

Decomposing Deep Neural Network Minds into Parts

by

Eric J. Michaud

B.A. Mathematics, UC Berkeley, 2021



Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN PHYSICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2026

© 2026 Eric J. Michaud. All rights reserved.

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: Eric J. Michaud
Department of Physics
October 29, 2025

Certified by: Max Tegmark
Professor of Physics, Thesis Supervisor

Accepted by: Lindley Winslow
Professor of Physics
Associate Department Head, Physics

THESIS COMMITTEE

THESIS SUPERVISOR

Max Tegmark
Professor of Physics
MIT

THESIS READERS

Isaac Chuang
Professor of EECS and Professor of Physics
MIT

Mike Williams
Professor of Physics
MIT

Decomposing Deep Neural Network Minds into Parts

by

Eric J. Michaud

Submitted to the Department of Physics
on October 29, 2025 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN PHYSICS

ABSTRACT

We are no longer alone in the universe. With the success of deep learning, we have new kinds of minds in the world that we can communicate with and study. This thesis is about understanding the internal workings of these new minds. Our approach is to decompose the computation that neural networks perform into parts—intelligible mechanisms that are more specialized and less complex than the network as a whole.

First, we study how the ways that neural networks learn and scale are influenced by the properties of these mechanisms. We study how the statistics of these mechanisms can produce “neural scaling laws”, how the relationship of these mechanisms to each other causes “curricula”, and how differences in the complexity of these mechanisms and the speed at which they are learned cause “grokking”.

We then study the challenge of automatically decomposing neural network computation into atomic parts. We first demonstrate that we can convert small networks into standalone Python programs. Turning to larger networks, we then study how sparse autoencoders can be used to decompose language model activations into features. We find multiple types of geometric structure to language model features, raising interesting questions about whether individual sparse autoencoder latents are the only sensible units of decomposition.

Thesis supervisor: Max Tegmark

Title: Professor of Physics

Acknowledgments

I am deeply grateful to so many colleagues, friends, and family members, without whom this PhD would not have been possible.

First, thank you Max Tegmark. Thank you, through your writing, for helping to inspire my interest in the subjects that became the focus of my PhD work, and for bringing me to MIT to do this work. I have fond memories of many chats, group meetings, and moments of curiosity and discovery together. Let us hope that our work will be used for good.

Thank you to my collaborators—I was unbelievably fortunate to work with you. Thank you Ziming Liu for being a generous collaborator and mentor during the early parts of my PhD, and for your creativity, humor, and camaraderie throughout this era of our lives. Thank you Josh Engels for your generosity, tenacity, good humor, and friendship during the time that we had together. Thank you Wes Gurnee for your support, energy, and friendship over the last few years. Thank you to David D. Baek for your camaraderie and kindness. Thank you Ouail Kitouni and Niklas Nolte for your friendship and encouragement early in my PhD. I was especially fortunate to work with many talented masters students and undergraduates too. Thank you Asher Parker-Sartori, Uzay Girit, Vedang Lad, Isaac Liao, Zifan (Carl) Guo, Xiaoqing (Lily) Sun, Anish Mudide, Chloe Loughridge, Tara Rezaei, Mateja Vukelic, and Xiaoman (Delores) Ding for your work and energy.

Thank you to Tom McGrath for inviting me to intern at Goodfire, and for your kindness and patience during my time there. Thank you too to Liv Gorton for your ideas, humor, and camaraderie.

Thank you to Ike Chuang, Jeff Gore, Mike Williams, David Bau, and Hidenori Tanaka for your guidance and mentorship during my PhD. Thank you to Shannon Larkin especially for your help with administrative matters.

I benefited from conversations and correspondence, about research and life, with many other colleagues and friends during my PhD. Just some of these people include Peter Slattery, William Brandon, Jamie Simon, Ekdeep Lubana Singh, Jack Merullo, Atticus Geiger, Rohin Shah, Irina Rish, Tony T. Wang, Stephen Casper, Chris Olah, Jesse Hoogland, Neel Nanda, Subhash Kantamneni, Daniel Kunin, Michael T. Pearce, Alex Infanger, Kaivu Hariharan, Eleni Shor, Fiona Pollack, Kuhan Jeyapragasan, J.J. Andrade, Adam Jermyn, and Jake Mendel.

I had wonderful mentors during my undergraduate studies who set me on the path to this PhD. Many also continued to provide mentorship and support during my PhD. From the Berkeley SETI Research Center, thank you Dave MacMahon, Howard Isaacson, Steve Croft, and Andrew Siemion. From the Lawrence Livermore National Laboratory, thank you Vic Castillo. From CHAI, thank you Adam Gleave. And thank you also to Erik Hoel.

I feel such gratitude for my friends. Thank you to Dan Bhatt, Tomer Fidelman, Nathan Wenger, Josh Fong, Tyler Brooks, David Collier, Ethan Christensen, Cristian Trout, Marvin von

Hagen, Davis Brown, Tamay Besiroglu, Trenton Bricken, Anthony Campolo and many others for supporting me through this period.

Thank you Grandpa (James Michaud), my namesake, for your love and gentle support for my interests throughout my life. I remember, as a small child, you taking me aside to explain how soap works and suggesting to me I should become a physicist. Well, I've done it!

Thank you Nana (Joanne Lewis Clever) for your love and belief in me throughout my life. I love you so very much.

Thank you Mom (Kimberly Clever Michaud) and Dad (David Edward Michaud) for your love and support from the beginning. Thank you for your effort in bringing me into this universe and for giving me a lovely home and start in life. Thank you for giving me an education, and the independence to pursue my interests throughout life. Thank you especially, Mom, for reading so much to me when I was young—I was excellently “pretrained”!

Contents

<i>List of Figures</i>	15
<i>List of Tables</i>	31
1 Introduction	33
1.1 The most interesting phenomenon in the universe	33
1.2 A brief history of deep learning	34
1.3 Deep learning at the microscale and macroscale	36
1.4 A “cognitive revolution” in the science of deep learning	37
1.5 The mesoscale: a physics of the “parts”	37
1.5.1 Mechanisms and their properties	37
1.5.2 What does this say about the world?	39
1.6 Contents and Contributions	40
2 The Quantization Model of Neural Scaling	47
2.1 Introduction	47
2.2 Theory	50
2.3 Proof of concept: a toy dataset	51
2.3.1 The “multitask sparse parity” dataset	52
2.3.2 Power law scaling and emergence	52
2.4 Decomposing LLM scaling laws	54
2.4.1 The distribution over per-token losses	54
2.4.2 Monogenic versus polygenic scaling curves	54
2.5 The quanta of language modeling	56
2.5.1 The natural distribution over language modeling quanta	58
2.6 Related Work	58
2.7 Discussion	59
2.8 Appendix	64
2.8.1 More general scaling laws	64
2.8.2 Additional results on multitask sparse parity	65
2.8.3 Additional results on language models	66
2.8.4 Quanta discovery on TinyStories	67
2.8.5 The difficulty of estimating the power law exponent from clusters	67
2.8.6 Parameter and data scaling exponents across studies	70
2.8.7 Estimates of compute used for our experiments	70

3	On the Creation of Narrow AI: Hierarchy and Nonlocality of Neural Network Skills	77
3.1	Introduction	78
3.2	Methods	80
3.3	Case study: compositional multitask sparse parity	80
3.3.1	Defining compositional multitask sparse parity (CMSP)	80
3.3.2	Learning dynamics on CMSP	81
3.3.3	Nonlocal representations in CMSP networks	83
3.3.4	Regularizing to “narrow” networks	85
3.4	Pruning vs distillation: MNIST	85
3.5	Pruning vs distillation: LLMs on Python documents	86
3.6	Related Work	89
3.7	Discussion	90
3.8	Appendix	96
3.8.1	Additional results on CMSP	96
3.8.2	Additional results on LLMs	97
3.8.3	Compute estimates	100
4	Precision Machine Learning	103
4.1	Introduction	104
4.1.1	Problem Setting	104
4.1.2	Decomposition of Loss	105
4.1.3	Importance of Scaling Exponents	106
4.1.4	Organization	106
4.2	Piecewise Linear Methods	107
4.3	Nonlinear Methods	111
4.4	Optimization	114
4.4.1	Properties of Loss Landscape	114
4.4.2	Optimization Tricks For Reducing Optimization Error	115
4.4.3	Limitations and Outlook	117
4.5	Conclusions	118
4.6	Appendix	122
4.6.1	Boosting advantage comes not just from rescaling loss	122
4.6.2	A closer look at low-curvature subspace optimization	122
4.6.3	Neural scaling of tanh networks with BFGS versus Adam optimizer	122
4.6.4	Loss landscape at lower loss	122
5	Towards Understanding Grokking: An Effective Theory of Representation Learning	127
5.1	Introduction	128
5.2	Problem Setting	129
5.3	Why Generalization: Representations and Dynamics	130
5.3.1	Representation quality predicts generalization for the toy model	130
5.3.2	The critical training set size	131
5.4	Delayed Generalization: A Phase Diagram	134
5.4.1	Phase diagram of a toy model	134
5.4.2	Beyond the toy model	136

5.5	Related work	137
5.6	Conclusion	138
5.7	Appendix	140
5.7.1	Applicability of our toy setting	140
5.7.2	Definition of $\widehat{\text{Acc}}$	141
5.7.3	The gap of a realistic model $\mathcal{M}$ and the ideal model $\mathcal{M}^*$	142
5.7.4	Conservation laws of the effective theory	145
5.7.5	More phase diagrams of the toy setup	146
5.7.6	General groups	148
6	Omnigrok: Grokking Beyond Algorithmic Data	153
6.1	Introduction	154
6.2	The LU mechanism for grokking	155
6.3	Grokking for a teacher-student setup	156
6.4	Omnigrok: Grokking for more interesting tasks	158
6.5	Representation is key to grokking	160
6.5.1	Algorithmic datasets	161
6.5.2	MNIST	164
6.6	Relation to Related works	165
6.7	Conclusions	165
6.8	Appendix	168
6.8.1	Experiment details	168
6.8.2	Reduced loss for modular addition with transformers	168
6.8.3	time to generalize versus weight decay	168
6.8.4	Section 6.5.1 setup	169
6.8.5	MNIST experiments with cross entropy loss	170
7	Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code	177
7.1	Introduction	178
7.2	Related Work	178
7.3	MIPS, our program synthesis algorithm	179
7.3.1	AutoML optimizing for simplicity	181
7.3.2	Auto-simplification	182
7.3.3	Boolean and integer autoencoders	183
7.4	Results	185
7.4.1	Benchmark	185
7.4.2	Evaluation	186
7.4.3	Performance	188
7.5	Discussion	191
7.5.1	Findings	191
7.5.2	Outlook	191
7.6	Appendix	195
7.6.1	Lattice finding using generalized greatest common divisor (GCD)	195
7.6.2	Problem formulation	195
7.6.3	Regular GCD	196

7.6.4	Generalized GCD in D dimensions	196
7.6.5	Linear lattice finder	198
7.6.6	Symbolic regression	199
7.6.7	Neural Network Normalization Algorithms	199
7.6.8	Supplementary training data details	203
7.6.9	Generated programs	203
7.6.10	Formal Verification	213
8	The Geometry of Concepts: Sparse Autoencoder Feature Structure	219
8.1	Introduction	220
8.2	Related work	221
8.3	“Atom” scale: crystal structure	222
8.4	“Brain” scale: Meso-scale modular structure	223
8.5	“Galaxy” scale: Large-scale point cloud structure	227
8.5.1	Shape analysis	228
8.5.2	Clustering analysis	229
8.6	Conclusion	230
8.7	Appendix	235
8.7.1	Additional Information on Brain Lobes	235
8.7.2	Understanding Principal Components in Difference Space	235
8.7.3	Breaking Down SAE Vectors by PCA Component	235
9	Not All Language Model Features Are One-Dimensionally Linear	239
9.1	Introduction	240
9.1.1	Contributions	240
9.2	Related Work	241
9.3	Definitions	242
9.3.1	Multi-Dimensional Features	242
9.3.2	Superposition	244
9.4	Sparse Autoencoders Find Multi-Dimensional Features	245
9.5	Circular Representations in Large Language Models	247
9.5.1	Intervening on Circular Day and Month Representations	248
9.5.2	Intervening with the SAE Plane	250
9.5.3	Continuity of Circular Representations	250
9.6	Discussion	251
9.7	Appendix	256
9.7.1	Multi-Dimensional Feature Capacity	256
9.7.2	More on Reducibility	258
9.7.3	Alternative Definitions	260
9.7.4	Toy Case of Training SAEs on Circles	262
9.7.5	Training Mistral SAEs	263
9.7.6	GPT-2 and Mistral 7B Dictionary Element Clustering	264
9.7.7	Other discovered clusters	267
9.7.8	Further Experiment Details	268
9.7.9	More Weekdays and Months Plots and Details	271

9.7.10	Patching	273
9.7.11	Explanation via Regression (EVR)	276
10	Understanding Sparse Autoencoder Scaling in the Presence of Feature Manifolds	281
10.1	Introduction	281
10.2	A model of sparse autoencoder scaling	282
10.2.1	The structure of data and the SAE architecture	282
10.2.2	Assumption: SAE optimization reduces to a latent allocation problem . . .	282
10.2.3	Warm-up on discrete features	283
10.2.4	Intuition behind pathological manifold scaling	284
10.2.5	Solution for power-law $L(n)$ following Brill (2024)	285
10.2.6	Numerical simulation	285
10.3	SAE scaling on synthetic features and on real neural networks	286
10.4	Discussion	286
10.5	Appendix	289
10.5.1	Additional discussion	289
10.5.2	Derivations	291
10.5.3	Additional Experimental Details	293
10.5.4	SAE scaling curves on synthetic manifolds	293
10.5.5	SAE feature geometry on LLMs and vision models	293

List of Figures

1.1	We suggest that the computation that neural networks perform can be decomposed into “parts” (mechanisms / modules). The chapters of this thesis address questions about the properties of these parts, and how we can automatically find decompositions of neural networks into parts.	38
2.1	We auto-discover <i>quanta</i> – basic units of model knowledge/skill – for a language model. Here we show collections of next-token prediction samples which our method clustered together, each corresponding to some coherent model behavior. We indicate the token which was predicted from the context before it with a red highlight. We indicate newlines using “\n”. See Section 2.5 for explanation. . . .	49
2.2	Top: Neural networks exhibit power law scaling in loss w.r.t. parameters N , training time S , and training samples D (for multi-epoch training) when trained on the multitask sparse parity dataset. Here $\alpha = 0.4$ and we plot lines $\propto N^{-\alpha}$, $\propto S^{-\alpha/(\alpha+1)}$, $\propto D^{-\alpha/(\alpha+1)}$. Bottom: neural scaling broken down by subtask. Scaling behavior on individual subtasks exhibits emergence, where subtasks are suddenly learned above a particular scale. Power law neural scaling of mean test loss averages over a large number of qualitative changes in network performance (when broken down by subtask), with loss being driven to zero on an increasing number of subtasks which are power law distributed in frequency, a realization of the mechanism of neural scaling discussed in Section 2.2.	53
2.3	Left: Scaling of mean test loss w.r.t. non-embedding parameters for the Pythia models [29]. The parameter scaling exponent α_N is measured to be ≈ 0.083 from the first six points along the curve (the seventh model appears to break the trend). Center: the distribution $p(L)$ over losses on individual samples for models of different size. Losses ≈ 0 are by far the most common, and larger models achieve ≈ 0 loss on an increasing fraction of samples. Right: the expected loss integrand $Lp(L)$ for models of different sizes. Low-loss samples contribute minimal mass to the mean loss, which is instead dominated by samples with much higher loss of 5-10 bits (depending on scale).	55

2.4	Per-sample scaling curves can have diverse behavior. Here we show extreme examples where scaling (of loss on predicting the highlighted token) is abrupt versus smooth. If the Quantization Hypothesis describes language modeling, then samples with sharp scaling would be <i>monogenic</i> , displaying sharp emergence at a particular model scale when the relevant quantum is learned. Samples with gradual scaling would be <i>polygenic</i> , where many quanta, emerging at different scales, marginally improve the loss. We show additional examples in Figure 2.12.	56
2.5	Left: angular similarity between model gradients for a variety of natural language samples. Samples are reordered according to their QDG cluster (with 400 clusters) to reveal the block-diagonal structure of the similarity matrix. We visualize a small part of the overall similarity matrix in this plot – note that not all clusters are as visibly distinct as the ones shown. Right: rank-frequency plot of QDG clusters. We measure the slope of the envelope of the rank-frequency curves from cluster rank 100-1000 to be ≈ -1.24 , which is a steeper than the slope of -1.08 expected from the measured parameter-scaling exponent from Figure 2.3, though within the margin of error given the uncertainty of our clustering methodology. See Section 2.8.5 for a discussion of the bias/uncertainty of our method.	57
2.6	Comparing different scaling laws. Setting $a_k = 0$, we compare $b_k = -\log p_k$ (solid lines) and $b_k = 1$ (dashed lines) for different alphas. Although the $b_k = -\log p_k$ case would cause an extra loss term $n^{-\alpha} \log n$ in addition to the power law term $n^{-\alpha}$, the loss becomes a power law asymptotically when n becomes large.	64
2.7	Training dynamics on the multitask sparse parity dataset consist of many “phase transitions” when decomposed by subtask – the loss curve for each subtask drops following an initial plateau of no apparent progress, in line with [28]. The mean loss decreases smoothly, averaging over these phase transitions in the model’s performance on subtasks. We show curves for single-epoch training (top) and multi-epoch training on 5 million samples (bottom). The dashed red line indicates the early stopping point where mean test loss is minimized. For these runs, $\alpha = 0.4$.	65
2.8	Convergence time for each subtask versus the frequency of that subtask. We see that convergence time S_k on subtask k is $S_k \propto p_k^{-0.81}$ rather than $S_k \propto p_k^{-1}$ as we had expected. This leads to a steeper scaling w.r.t. S than expected from theory. For these experiments, we used $\alpha = 0.4$, and so we would have predicted $\alpha_S \approx 0.29$ but instead we get $\alpha_S \approx 0.45$. We consider the model to have converged on a subtask once it gets mean test loss less than 0.1 bits on that subtask.	66
2.9	Number of subtasks learned (n), including subtasks learned after early-stopping would have terminated the training run, versus training samples D for a variety of α . We see that the relation $n \propto D^{1/(\alpha+1)}$ approximately holds, in line with theory. Deviation from theory for the scaling exponent of loss L w.r.t. D therefore likely originates from our failure to regularize network training, leading to early-stopping ending training before some subtasks can be learned.	67
2.10	Scaling in parameters (N), single-epoch training time (S), and multi-epoch training samples (D) for varying quanta power law distribution parameter α on multitask sparse parity. We notice that scaling curves in steps S are typically steeper than the $\alpha_S = \alpha/(\alpha + 1)$ predicted from theory, and that for low α the scaling curves in D also deviate from theory substantially.	68

2.11	Left: Training curves (scaling w.r.t. steps S) of mean test loss for Pythia models. We measure exponents α_S between 0.037 and 0.06. Center: the distribution $p(L)$ over time. Over time, models achieve ≈ 0 loss on an increasing fraction of tokens, similar to scaling in model size. Right: The distribution $L \cdot p(L)$ over time.	69
2.12	Additional LLM scaling curves on individual samples which exhibit sharp vs smooth improvement. If the Quantization Hypothesis is true for language modeling, then we would interpret samples with sharp drops as “monogenic” and samples with gradual progress as “polygenic”.	71
2.13	Additional examples of clusters of inputs discovered by QDG. Like in Figure 2.1, we used 10000 samples and <code>n_clusters</code> of 400.	72
2.14	Examples of clusters within the TinyStories dataset, discovered by QDG on the TinyStories-33M model. Here we just show samples from four out of 400 <code>n_clusters</code>	73
2.15	Similarity matrix and rank-frequency plots from QDG on multitask sparse parity. Despite sparse parity having a known decomposition into subtasks which are power law distributed in frequency, we do not recover this same power law from samples. We used $\alpha = 0.4$ for the frequency distribution for an expected rank-frequency power law exponent of -1.4, but measure a rank-frequency envelope slope closer to -1.1.	74
2.16	To understand the bias of spectral clustering, we apply spectral clustering to a toy model with different embedding dimension d , noise scale σ and number of cluster k . The high-dimension ($d = 1000$) large-noise ($\sigma = 2.0$) scheme seems to best agree with the LLM results (Figure 2.5).	74
2.17	The difficulty of measuring α from curves. We apply spectral clustering to a toy model with different α and number of clusters k . For a fixed α , different k curves define an envelope. One could use the envelope slope to infer α , but this incurs errors around 0.2.	75
2.18	Parameter and data scaling exponents from various studies of deep learning scaling, compiled from the database of neural scaling laws from [47]. Our model of scaling predicts that $\alpha_D = \alpha_N / (\alpha_N + 1)$, indicated with the solid black line. Visible points are from [2,3,5,7,48]. [49] is above the visible window of the figure.	75
3.1	We study two challenges to making strong, narrow-purpose AI models. (A): Data may have <i>hierarchical</i> structure. If skills have a hierarchical dependence, where some skills are only learnable after more primitive skills are learned first, then it sometimes may be necessary to train on a broad distribution of data to learn certain narrow skills within that distribution. These dynamics may mean that general-purpose models must be trained to achieve certain performance on some domains. (B): Model features are <i>distributed</i> . By default, skills may not be localizable to a particular set of model components (e.g. neurons). In this case, pruning of model components won’t precisely retain wanted skills and remove unwanted skills from models. We explore methods for aligning the model features relevant to particular domains with a smaller subset of model components while unlearning others. . .	79

3.2	Training dynamics on compositional multitask sparse parity. Top: training dynamics for a single network trained on four atomic subtasks $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, and their composition $\{0,1,2,3\}$. Bottom left: loss on compositional subtask $\{0,1,2,3\}$, when training on atomic subtasks and their composition, as in the top subplot, across 40 network seeds (depth 3, width 128). Bottom right: loss on compositional subtask $\{0,1,2,3\}$, when only training on samples from that composite task, <i>without</i> also training on atomic tasks, across 40 network seeds (depth 3, width 128). We find that removing the atomic tasks prevents our networks from learning the composite task. We report the minimum loss within the previous 100 steps of training to filter out loss spikes.	82
3.3	Top: We visualize the connectivity of 2-hidden-layer MLPs trained on a CMSP distribution with subtasks $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1,2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{3,4,5\}$, visualized before (left) and after (right) regularizing network weights with the group lasso sparsity penalty while training on subtask $\{0,1,2\}$. We find that network connectivity becomes sparse after regularizing. Negative weights are shown in blue and positive ones in red, with width proportional to the norm of the weight. Bottom: we show how pruning affects task performance on subtasks $\{0,1,2\}$ and $\{3,4,5\}$ at varying sparsity levels. We prune neurons based on the absolute change that ablating them has on the loss on subtask $\{0,1,2\}$. We find that subtasks here are nonlocal and entangled in the “pretrained” network (left). As we prune neurons according to their relevance on subtask $\{0,1,2\}$, at the sparsity at which performance on subtask $\{0,1,2\}$ accuracy drops below 98% (green line), we can still recover some performance on subtask $\{3,4,5\}$ with a small amount of additional training. Thus naive pruning here has not completely and robustly unlearned subtask $\{3,4,5\}$. However, after regularizing the weights (right), we find that not only we can more aggressively prune the network, but we have also robustly unlearned subtask $\{3,4,5\}$. Note that the degree to which subtasks are nonlocal and entangled in the pretrained networks depends on seed and width, and we show a variety of additional curves in Figure 3.10.	84
3.4	Left: We compare the performance of distillation, training from scratch, and two pruning approaches for creating small networks that classify MNIST even digits. Pruning-based approaches Pareto-dominate distillation, achieving high compression ratios with fewer datapoints. All points are averaged over 10 individual training runs. Right: When pruning using group lasso, it often helps to first prune rapidly, degrading performance, and then recover performance with no regularization. Each line represents a single training run, with a new point logged every 2,000 datapoints. Darker lines correspond to more aggressive pruning runs that attain lower final neuron counts.	86

3.5	Left: Neuron sparsity vs. loss curve for LLMs finetuned with group lasso regularization with varying λ for 70k steps, as compared to the base network. Regularization flattens the sparsity vs. loss curve, at the cost of slightly degrading model performance. Right: After pruning our networks to 30%, 63%, and 80% sparsity for our runs with λ of 5e-4, 3e-4, and 1e-3, respectively, we recover performance with additional training. We find that we can recover performance lost during pruning, including in the network that was pruned without first using group-lasso regularization training. The plot shows an exponential moving average of batch losses, with individual batch losses shaded in back.	87
3.6	Frontier of network parameters vs. data required to achieve a certain cross-entropy loss on Python documents. For the task of creating a LLM specialized on Python documents, we find that pruning Llama-3.2-1B and then performing recovery training is much more efficient than training LLMs from scratch or distilling LLMs from scratch on the soft targets of Llama-3.2-8B.	88
3.7	Impact of network shape on CMSP learning. Top: Forty runs of training with different seeds on CMSP dataset $\mathcal{D}_{\{0\}} \cup \mathcal{D}_{\{1\}} \cup \mathcal{D}_{\{2\}} \cup \mathcal{D}_{\{3\}} \cup \mathcal{D}_{\{0,1,2,3\}}$ with $m = 4$, $n = 64$, $k = 4$. We find that 27/40 runs converge with two hidden layers versus 19/40 runs with a single hidden layer with width 128 and 16/40 runs with a single hidden layer and width 361. Single-hidden-layer networks are therefore able to learn composite subtasks in CMSP roughly as well as deeper networks. Bottom: Forty runs of training on a standalone atomic task $m = 1$, $n = 64$, $k = 4$	96
3.8	For each hidden neuron in the first hidden layer (left) and the second hidden layer (right) in a CMSP network, we show the ablation score of that neuron for each subtask. For some neurons, ablation scores are high across multiple subtasks, even from separate “skill trees”.	97
3.9	Ablation scores for each subtask and each neuron. This is simply a different way of visualizing the results in Figure 3.8.	97
3.10	Pruning sparsity vs accuracy curves for pretrained CMSP networks. We train MLPs of varying width and random seed on a CMSP dataset with two skill trees, as in Figure 3.3. We prune neurons based on ablation score on subtask $\{0,1,2\}$. On some networks, ablating neurons to maximum sparsity while preserving performance on subtask $\{0,1,2\}$ robustly unlearns subtask $\{3,4,5\}$. However, for other networks, the subtasks seem more entangled, and we can recover performance on subtask $\{3,4,5\}$ even after ablating to the maximum extent we can still recover 98% accuracy on subtask $\{0,1,2\}$ (green line).	98
3.11	Comparison of ablation vs. attribution scores for Llama-3.2-1B neurons (left) and residual stream dimensions (right), evaluated on a single batch of Python code documents. For each figure, we fully ablate model components and compare the absolute change in loss (Abs Ablation Score, simply called “ablation score” elsewhere in the paper) with the absolute value of the linear estimate of this change computed from model gradients (Abs Attribution Score, called “attribution score” elsewhere). Note that the correlation between the log of these scores is 0.80 for neurons and 0.05 for residual stream components.	99

3.12	Training curves of LLMs on Python documents when training from scratch (left), training from scratch but with distillation of a larger pretrained LLM (center), and when training pruned pretrained models (right) to recover performance lost during training.	100
3.13	Recovery training curves after pruning neurons and residual stream dimensions, for Llama-3.2-1B on Python code documents. We compare recovery performance when pruning based on attribution scores vs. choosing components randomly. Attribution scores are computed across 1024 documents with a max length of 1024 tokens. We find that pruning with attribution scores is better than pruning random components, however this gap is eventually recovered. For instance, at step 5365 when our run with 50% neuron sparsity and 20% residual sparsity first matches the performance of the original model (≈ 1.3 nats), the performance on randomly-pruned model is almost identical at 1.301 nats.	101
4.1	In (a) (top), we show the solutions learned by a ReLU network and linear simplex interpolation on the 1D problem $y = \cos(2x)$. In (b) (bottom), we visualize linear regions for a ReLU network, trained on unnormalized data (left) and normalized data (center), as well as linear simplex interpolation (right) on the 2D problem $z = xy$. In general, we find that normalizing data to have zero mean and unit variance improves network performance, but that linear simplex interpolation outperforms neural networks on low-dimensional problems by better vertex placement.	107
4.2	Scaling of linear simplex interpolation versus ReLU NNs. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, we find that NNs sometimes scale better (at least in early regimes) as N^{-2/d^*} , where $d^* = 2$, on high dimensional problems.	108
4.3	ReLU neural networks are seen to initially scale roughly as if they were modular. Networks with enforced modularity (dark blue and red, dashed line), with architecture depicted on the right, perform and scale similarly, though slightly better, than standard dense MLPs of the same depth (light blue and red).	109
4.4	Interpolation methods, both linear and nonlinear, on 2D and 3D problems, seen to approximately scale as $D^{-(n+1)/d}$ where n is the order of the polynomial spline, d is the input dimension.	112
4.5	Scaling of linear simplex interpolation vs tanh NNs. We also plot ReLU NN performance as a dotted line for comparison. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, tanh NN scaling is much messier.	113
4.6	(a) Scaling of neural networks on a target function which can be arbitrarily closely approximated by a network of finite width. (b) diagram from [36] showing how a 4-neuron network can implement multiplication arbitrarily well. Therefore a depth-2 network of width at least 12 has an architecture error at the machine precision limit, yet optimization in practice does not discover solutions within at least 10 orders of magnitude of the precision limit.	114

4.7	Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training with the Adam optimizer, along with the magnitude of the gradient's projection onto each corresponding eigenvector (thin red line). We see a cluster of top eigenvalues and a bulk of near-zero eigenvalues. The gradient (thin jagged red curve) points mostly in directions of high-curvature.	115
4.8	Comparison of Adam with BFGS + low-curvature subspace training + boosting. Using second-order methods like BFGS, but especially using boosting, leads to an improvement of many orders of magnitude over just training with Adam. Target functions are a teacher network (top) and a symbolic equation (bottom).	116
4.9	Comparison of Adam with BFGS + low-curvature subspace training + boosting, for a 2D problem (top) and a 6D problem (bottom), the equation we studied in Figure 4.6(a). As we increase dimension, the optimization tricks we tried in this work show diminishing benefits.	117
4.10	User's Guide for Precision: which approximation is best depends on properties of the problem.	119
4.11	Rescaling loss has minimal benefit relative to boosting.	123
4.12	Low-curvature subspace training curves for varying thresholds τ	124
4.13	Scaling of tanh networks with the BFGS vs Adam optimizer. We use the same setup as in Figure 4.5, training tanh MLPs of depth 2-4 of varying width on functions given by symbolic equations. BFGS outperforms Adam on the 3-dimensional example shown (top left) and performs roughly similarly to Adam on the other problems.	124
4.14	Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training a width-20, depth-3 network to fit $y = x^2$ with the BFGS optimizer. Like in Figure 4.7, we also plot the magnitude of the gradient's projection onto each corresponding eigenvector (thin red line). The "canyon" shape of the loss landscape is more obvious at lower-loss points in the landscape found by BFGS than Adam finds. There is a clear set of top eigenvalues corresponding to a few directions of much higher curvature than the bulk.	125
5.1	Visualization of the first two principal components of the learned input embeddings at different training stages of a transformer learning modular addition. We observe that generalization coincides with the emergence of structure in the embeddings. See Section 5.4.2 for the training details.	129
5.2	Visualization of the learned set of embeddings ($p = 11$) and the decoder function associated with it for the case of 2D embeddings. Axes refer to each dimension of the learned embeddings. The decoder is evaluated on a grid of points in embedding-space and the color at each point represents the highest probability class. For visualization purposes, the decoder is trained on inputs of the form $(\mathbf{E}_i + \mathbf{E}_j)/2$. One can read off the output of the decoder when fed the operation $i \circ j$ from this figure simply by taking the midpoint between the respective embeddings of i and j	131
5.3	We compute accuracy (of the <u>full</u> dataset) either measured empirically Acc , or predicted from representation $\widehat{\text{Acc}}$. These two accuracies as a function of training data fraction are plotted in (a)(b), and their agreement is shown in (c).	132

5.4	(a) The effective theory predicts a phase transition in the probability of obtaining a linear representation around $r_c = 0.4$. (b) Empirical results display a phase transition of RQI around $r_c = 0.4$, in agreement with the theory (the blue line shows the median of multiple random seeds). The evolution of 1D representations predicted by the effective theory or obtained from neural network training (shown in (c) and (d) respectively) agree creditably well.	133
5.5	Phase diagrams of learning for the addition group and the permutation group. (a) shows the competition between representation and decoder. (b)(c)(d): each phase diagram contains four phases: comprehension, grokking, memorization and confusion, defined in Table 5.1. Grokking is sandwiched between comprehension and memorization.	135
5.6	Left: Evolution of the effective dimension of the embeddings (defined as the exponential of the entropy) during training and evaluated over 100 seeds. Center: Effect of dropout on speeding up generalization. Right: Phase diagram of the transformer architecture. A scan is performed over the weight decay and learning rate of the decoder while the learning rate of the embeddings is kept fixed at 10^{-3} (with zero weight decay).	137
5.7	As we include more data in the training set, the (ideal) model have a more structured representation, from (a) to (b) to (c).	141
5.8	We compare RQI and Acc for an ideal algorithm (with bar) and a realistic algorithm (without bar). In (a)(b)(d)(e), four quantities (RQI, $\overline{\text{RQI}}$, Acc, $\overline{\text{Acc}}$) as functions of training data fraction are shown. In (c)(f), RQI and Acc of the ideal algorithm sets upper bounds for those of the realistic algorithm.	143
5.9	$p = 4$ case. Equation set A (or geometrically, representation) has a hierarchy: $a \rightarrow b$ means a is a parent of b , and b is a child of a . A realistic model can only generate representations that are descendants of the representation generated by an ideal model.	144
5.10	$p = 4$ case. Representations obtained from training neural networks are displayed. η_1 and η_2 are learning rates of the representation and the decoder, respectively. As described in the main text, $(\eta_1, \eta_2) = (10^{-2}, 10^{-3})$ (right) is more ideal than $(\eta_1, \eta_2) = (10^{-3}, 10^{-2})$ (left), thus producing representations containing more parallelograms.	145
5.11	Phase diagrams of decoder learning rate (x axis) and batch size (y axis) for the addition group (left: regression; right: classification). Small decoder learning rate and large batch size (bottom left) lead to comprehension.	147
5.12	Phase diagrams of decoder learning rate (x axis) and initialization (y axis) for the addition group (left: regression; right: classification). Small initialization scale (top) leads to comprehension.	147
5.13	Phase diagrams of decoder learning rate (x axis) and representation weight decay (y axis) for the addition group (left: regression; right: classification). Representation weight decay does not affect model performance much.	148
5.14	Deduction of parallelograms	149
5.15	Permutation group S_3 . First three principal components of six embedding matrices $\mathbb{R}^{3 \times 3}$	150

5.16	Permutation group S_3 . (a) RQI increases as training set becomes larger. Each scatter point is a random seed, and the blue line is the highest RQI obtained with a fixed training set ratio; (b) steps to reach $RQI > 0.95$. The blue line is the smallest number of steps required. There is a phase transition around $r_c = 0.5$. (c) real accuracy Acc ; (d) predicted accuracy $\hat{\text{Acc}}$; (e) comparison of Acc and $\hat{\text{Acc}}$: $\hat{\text{Acc}}$ serves as a lower bound of Acc	151
6.1	(a) w : L_2 norm of model weights. Generalizing solutions (green stars) are concentrated around a sphere in the weight space where $w \approx w_c$ (green). Overfitting solutions (orange) populate the $w \gtrsim w_c$ region. (b) The training loss (orange) and test loss (gray) have the shape of L and U, respectively. Their mismatch in the $w > w_c$ region leads to fast-slow dynamics, resulting in grokking.	155
6.2	Teacher-student setup. α : student initialization scale, γ : weight decay. (a) The reduced training loss and test loss have the shape of "L" and "U", respectively. (b) Top row: large initialization ($\alpha = 2.0$) can demonstrate no generalization (no reg), grokking (small reg) and fast generalization (large reg). Bottom: small initialization ($\alpha = 0.5$) always generalizes fast, regardless of weight decay. (c) $\alpha = 2$. The steps to overfitting is independent of weight decay, while the steps to generalization scale inversely with the weight decay.	157
6.3	MNIST. (a) reduced training error, (b) reduced test error. Comparing A and B: larger weight norm makes learning grok (delay generalization). Comparing B and C: a larger training data size makes learning de-grok (speed up generalization). (c) "LU" holds truer for smaller data. (d) Accuracy curves for MNIST in the setting where we observe grokking. (e) Time to generalize as a function of training set size N	158
6.4	We use an LSTM to predict IMDb reviews. (a) training error; (b) test error; (c) reduced losses for data size 1k (top) and 50k (bottom); (d) With 1k data, a (weak) grokking signal is observed for large initializations ($\alpha = 6$), while no grokking is observed for standard initializations ($\alpha = 1$).	159
6.5	We use a GCNN to predict isotropic polarizability of molecules in the QM9 dataset. (a) training loss; (b) test loss; (c) reduced losses for data size 100 (top) and 3000 (bottom); (d) with 200 training samples, grokking is observed for large initialization ($\alpha = 3$), while no grokking is observed for standard initializations ($\alpha = 1$).	159
6.6	Loss landscapes for a toy MLP, on the 2D (w, m) plane. (a) Training loss splits the plane into two regions: large loss small w (fast dynamics) and small loss large w (slow dynamics). (b) Test loss; the green star is the generalizing solution. (c) Losses along an illustrative path $A \rightarrow E$, demonstrating multiple descent; (d) zoom-in of the training loss highlighting the gradients on the boundary. (e) the boundary depends on training data size; (f) a simple illustration of grokking dynamics. . . .	162
6.7	Training 1L transformer on modular addition ($p = 113$). (a) Weight norm, train accuracy, and test accuracy over time, initialized and trained normally. Weight norm first increases, and is highest during the period of overfitting, but then drops to become lower than initial weight norm when the model generalizes. (b) Constrained optimization at constant weight norm ($\alpha = 0.8$) largely eliminates grokking, with test and train accuracy improving concurrently.	163

6.8	MNIST landscapes as functions of representation messiness m and weight norm w : (a) training loss, and (b) test loss. Training and test losses do not have significant mismatch, and neither of them depend on representation strongly, which is in stark contrast to algorithmic datasets (Figure 6.6). (c) an illustrative path $A \rightarrow B \rightarrow C$ does not manifest grokking.	164
6.9	Reduced loss landscapes for transformers trained on modular addition, the original setting where grokking was observed.	169
6.10	Time to generalize as a function of weight decay: we investigate to what extent the relation $t \propto \gamma^{-1}$ holds, where t is number of training steps needed for the model to generalize and γ is the AdamW weight decay. When a lower weight decay is used, models spend longer in the period of overfitting before eventually generalizing. We show the generalization time t as a function of γ in (a)-(b) and full training curves for these runs in (c)-(d).	172
6.11	MNIST with the cross entropy loss (as opposed to the MSE loss used in Figure 6.3). (a) reduced training error, (b) reduced test error. (c) "LU" still holds for the cross entropy loss, but the effect is milder than the MSE loss. In particular, the "Goldilocks zone" (the weight range where generalization happens) is broader.	173
6.12	Training curves using cross entropy loss on MNIST. We are still able to observe delayed generalization on MNIST using cross entropy loss, though test accuracy first plateaus at higher than random-guess accuracy.	173
7.1	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.	180
7.2	These hidden structures can be turned into discrete representations. Left: bitstring addition, corresponding to 2 bits: the output bit and the carry bit. Right: Sum_Last2 task, 2D lattice corresponding to two integers.	183
7.3	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.	188
7.4	Comparison of code generated from an RNN trained on Sum_Last5, without (top) and with (bottom) normalizers.	189
7.5	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.	190
7.6	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).	198

8.1	Parallelogram and trapezoid structure is revealed (left) when distractor dimensions were projected out from the activations using LDA. LDA results in tighter clusters of pairwise gemma-2-2b activation differences (right), where each cluster corresponds to different semantic transformation. Distractor features are defined as those that are not related to semantics of the text; for instance, the first principal component of gemma-2-2b’s layer 0 activations (top-left figure on the right panel) represents word length. Parallelogram or trapezoid structures suggest that there is a unique direction in the activation space that represents each semantic transformation.	222
8.2	Features in the SAE feature point cloud identified that tend to fire together within documents are seen to also be geometrically co-located in functional “lobes”, here down-projected to 2D with t-SNE with point size proportional to feature frequency. A 2-lobe partition (left) is seen to break the point cloud into roughly equal parts, active on code/math documents and English language documents, respectively. A 3-lobe partition (right) is seen to mainly subdivide the English lobe into a part for short messages and dialogue (e.g. chat rooms and parliament proceedings) and one primarily containing long-form scientific papers.	224
8.3	Comparison of the lobe partitions of the SAE point cloud discovered with different affinity measures, with the same t-SNE projection as Figure 8.2. In the top left, we show clusters computed from geometry , the cosine similarity between features as the affinity score for spectral clustering. All other measures are based on whether SAE features co-occur (fire together) within 256-token blocks, using different measures of affinity. Although the Phi coefficient predicts spatial structure best, all co-occurrence measures are seen to discover the “code/math lobe”.	225
8.4	Top Left: Adjusted mutual information between spatial clusters and functional (cooccurrence-based) clusters. Top Right: logistic regression balanced test accuracy, predicting cooccurrence-based cluster label from position. Bottom Left: Adjusted mutual information with randomly-permuted cosine similarity-based clustering labels. Bottom Right: balanced test accuracy with random unit-norm feature vectors. The statistical significance reported is for Phi-based clustering into lobes.	226
8.5	Fraction of contexts in which each lobe had the highest proportion of activating features. For each document type, these fractions sum to 1 across the lobes. We see that lobe 2 typically disproportionately activates on code and math documents. Lobe 0 and 1 activate on other documents, with lobe 0 activating more on documents containing short text and dialogue (chat comments, parliamentary proceedings) and lobe 1 activating more on scientific papers.	227
8.6	3D Point Cloud visualizations of top PCA components for the gemma2-2b layer 12 SAE features.	228
8.7	Eigenvalue distributions for SAE features and activations. Eigenvalues of the covariance matrix for SAE features (left) and neural activations (right) decay approximately as a power law, with slopes varying across layers. A scaled isotropic Gaussian spectrum is shown for comparison, highlighting the significantly steeper decay for SAE features. Eigenvalue spectra for activations show a much slower decay compared to SAE features, indicating weaker power-law behavior and distinct geometric structures.	229

8.8	Layer-Wise Analysis of Latent Representations. Left: The power-law slope (α) of the eigenvalue spectrum (blue) and the log-determinant of the covariance matrix (green) vary across layers. Both metrics peak in intermediate layers, indicating significant structural changes in the latent space. Right: Estimated clustering entropy across layers with 95% confidence intervals. Middle layers exhibit reduced clustering entropy, while earlier and later layers show higher entropy, reflecting distributed and concentrated feature representations, respectively.	229
8.9	Histogram, over all features, of Phi coefficient with k -th nearest cosine similarity neighbor for (left) Layer 0 and (right) Layer 12. Each line represents a different k . The “random” line is plotted by drawing a random feature for each feature, then computing the Phi coefficient. Features with higher cosine similarity have higher Phi coefficient, but this is less pronounced in layer 0 compared to layer 12.	236
8.10	Plot of the first principal component in the difference space as a function of last token length difference in gemma-2-2b layer 0. The linear relationship indicates that the first principal component encodes the length difference between two words’ last tokens.	236
8.11	Silhouette Score, a measure of clustering quality, as a function of reduced dimension in LDA. The plot indicates that training an affine transformation for semantic cluster separation is easier in mid-layer (layer 12), where the model starts to develop concept-level understanding of the input.	237
8.12	Smoothed PCA scores for each SAE feature of the layer 12, width $16k$, $L_0 = 176$ Gemma Scope 2b SAE, sorted by frequency. PCA score = $\frac{1}{n} \sum_i i * (pca_i @ feature_j)^2$, where n is the number of PCA features. The smoothed curves just average this somewhat noisy metric over adjacent sorted features. This measures approximately where in the PCA each SAE feature lies, and shows that there is a dip into earlier PCA features in the last third of SAE features.	238
9.1	Circular representations of days of the week, months of the year, and years of the 20th century in layer 7 of GPT-2-small colored by the token they fire on. These representations were discovered via clustering SAE dictionary elements, described in Section 9.4. Points are colored according to the token which created the representation. See Figure 9.14 for other axes and Figure 9.15 for similar plots for Mistral 7B.	241
9.2	Empirical ϵ -mixture index and separability index for the “days of the week” cluster along PCA components 2 and 3. Left: The ϵ band parameterized by $\mathbf{v}$ and c that the optimization procedure found contained the highest fraction of points. Mid: Dot products of points in the feature distribution with the ϵ band; $M_\epsilon(\mathbf{f})$ is the percent of dot products within $\epsilon = 0.1$ of 0. Right: Estimated mutual information for different rotations of the space; $S(\mathbf{f})$ is the minimum over all rotations. This point cloud has a lower ϵ -mixture index and higher separability index than PCA projections within typical clusters (see fig. 9.3), indicating that it is more likely to be an irreducible multi-dimensional feature.	245
9.3	Mixture index and separability index of GPT-2 features. Features from fig. 9.1, which we had manually identified, score highly as candidate multidimensional features with these metrics.	246

9.4	Top two PCA components on the α token. Colors show α . Left: Layer 30 of Mistral on Weekdays. Right: Layer 5 of Llama on Months.	247
9.5	Visual representation of the circular intervention process. Top: We learn a circular probe on the PCA projection of a training set. Bot: To intervene, we change the circular representation to α'_j and average ablate other dimensions.	248
9.6	Mean and 96% error bars for intervening on the α token across layers using different intervention methods. The circular intervention technique outperforms patching only the top 5 PCA components and leaving the rest unchanged, and almost reaches the upper bound performance of patching the entire layer.	248
9.7	Off distribution interventions on Mistral layer 5 on the Weekdays task. The color corresponds to the highest logit γ after performing the circular subspace intervention on that point.	249
9.8	Interventions on the Mistral 7B Weekdays task with different methods of determining the probe.	250
9.9	Layer 30 Mistral 7B activations for [morning/evening] on [Monday/Tuesday/.../Sunday], plotted projected into the PCA plane for [Monday/Tuesday/.../Sunday].	251
9.10	Testing irreducibility of synthetic features. Left in each subfigure: Distributions of $\mathbf{x}$. For feature a , 63.96% lies within the narrow dotted lines, indicating the feature is likely a mixture. For feature b , 17.84% lies within the wide lines, indicating the feature is unlikely to be a mixture. The green cross indicates the angle θ that minimizes mutual information. Middle 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: Mutual information between a and b as a function of the rotation angle θ of matrix R . Feature b has a large minimum mutual information so is unlikely to be separable; feature a has a medium value of minimum mutual information of about 0.37 bits.	260
9.11	Testing irreducibility of synthetic features. Left in each subfigure: Distributions of $\mathbf{x}$. For feature c , 7.94% lies within the narrow dotted lines, indicating the feature is unlikely to be a mixture. For feature d , 25.90% lies within the wide lines, indicating the feature is likely a mixture. The green cross indicates the angle θ that minimizes mutual information. Middle 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: Mutual information between a and b as a function of the rotation angle θ of matrix R . Both features have a small (< 0.5 bits) minimum mutual information and so are likely separable.	261
9.12	SAEs trained to reconstruct a single 2d circle with $m = 2$ (left) and $m = 10$ (middle and right) dictionary elements. When there are several SAE features, there is no natural/canonical choice of feature directions, and the dictionary elements spread out across the circle.	263
9.13	Automatic discovery of synthetic circular features by clustering SAE dictionary elements.	264
9.14	Projections of days of week, months of year, and years of the 20th century representations onto top four principal components, showing additional dimensions of the representations than fig. 9.1.	266

9.15	Circular representations of days of the week and months of the year which we discover with our unsupervised SAE clustering method in Mistral 7B. Unlike similar features in GPT-2, we also find an additional “weekend” representation in between Saturday and Sunday representations (left) and additional representations of seasons among the months (right). For instance, “winter” tokens activate a region of the circle in between the representation of January and December. . . .	267
9.16	Hyperparameter regimes where the days of the week cluster exists. The cluster exists in the regime between all features clumping together and all features being in their own cluster; this regime seems reasonably stable.	268
9.17	Top 10 GPT-2 clusters by Mixture and Separability Index.	270
9.18	Projections onto the top two PCA dimensions of model hidden states on the α token show that circular representations of α are present in various layers. . . .	271
9.19	Attention and MLP patching results on Weekdays. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α	272
9.21	Projections of Mistral 7B Weekdays task activations at layer 8 into the plane discovered by clustering layer 8 SAE features.	272
9.20	Attention and MLP patching results on Months. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α	273
9.22	Layer 30 Mistral 7B activations for [very early/very late] on [Monday/Tuesday/.../Sunday], plotted projected into the PCA plane for [Monday/Tuesday/.../Sunday].	273
9.23	Top two PCA components of residual errors after EVR with one-hot in α and β . Mistral 7B Weekdays, layer 25, final token. Colored by γ	276
9.24	EVR residual RGB plots on Mistral hidden states on the Weekdays final token, layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) $\mathbf{g}_i$ labelled just underneath, as well as the resulting r^2 value. We write “tmr” meaning “tomorrow” for $\beta = 1$. We also write “circle for x ” meaning the inclusion of two functions $\mathbf{g}_i(x) = \{\cos, \sin\}(2\pi x/7)$	277
9.25	Iterative deconstruction of hidden state representations on the final token on Llama 3 8B, Months.	279
10.1	SAE scaling on a toy feature manifold S^1. We train ReLU SAEs with an L1 sparsity penalty to reconstruct points on the circle $S^1 \subseteq \mathbb{R}^2$. We find that SAEs can slightly reduce their total loss by “tiling” the manifold with more sparsely activating latents. For SAEs with 4, 8, and 24 latents, we show the data (S^1) in grey, the SAE’s reconstruction in black, the decoder latent directions as arrows, and indicate the part of the circle that each latent fires on as colored arcs. If SAEs can reduce loss by “tiling” a manifold in this way, they may do this at the expense of learning rarer features.	283
10.2	Left: Application of Brill’s (2024) [25] capacity-allocation model to SAE scaling. Right: numerical simulation of SAE scaling when the most frequently occurring feature is a manifold with loss scaling as $L(n) \propto n^{-\beta}$ and all other features are discrete. We see that if $\beta \ll \alpha$, then a simulated SAE with 100 million latents discovers only 3 million features.	284

10.3	ReLU SAE scaling on individual toy feature manifolds, showing how $L(n)$ curves depend on feature geometry. We train with L1 sparsity $\lambda = 0.1$. Left: SAE scaling on data drawn from a single unit hypersphere. Right: SAE scaling on data sampled in $\{\mathbf{x} : 0.5 < \mathbf{x} < 2\}$	285
10.4	Frequencies at which latents fire in gemma scope SAEs, sorted by frequency. We measure the power law decay exponent between latent 10^2 and 10^4 to be -0.74 for an SAE with 16k latents and -0.57 for an SAE with 32k latents.	290
10.5	Measured slopes of L_i curves for ReLU L1 SAEs trained on hyperspheres.	291
10.6	JumpReLU SAE scaling on individual toy feature manifolds, showing how $L(n)$ curves depend on feature geometry. We train with the tanh loss from [35] with $c = 0.1$ and $\lambda_s = 1.0$. Left: JumpReLU SAE scaling on unit hyperspheres of varying dimension. Right: JumpReLU SAE scaling on points sampled in $\{\mathbf{x} : 0.5 < \mathbf{x} < 2\}$. We see that, like with ReLU SAEs, that when there is variation in the radial direction between samples that our SAEs do not learn solutions which can continue to accommodate latents, and instead plateau after allocating roughly $2d_i$ latents to the feature manifold.	294
10.7	Distribution over cosine similarities between decoder vectors and their nearest neighbor.	294
10.8	Distribution over pairwise cosine similarities for Inception V1.	295

List of Tables

3.1	Unlearning: We prune Llama-3.2-1B using various techniques to create narrow Python-specific models, then evaluate on other tasks to assess whether pruning led to unlearning. Accuracies are reported for the pruned model before finetuning and after finetuning on a train set of each benchmark (non-FT→FT). ✓ indicates successful unlearning (FT score $\geq 10\%$ lower than base).	89
3.2	Transformer model configurations explored	99
3.3	Neuron and residual sparsity configurations when using attribution pruning on Llama-3.2-1B.	100
5.1	Definitions of the four phases of learning	136
6.1	Threes setups used in this paper, with different set of parameters trainable.	170
7.1	Benchmark results. For tasks with note “see text”, please refer to Appendix 7.6.8. The last column highlights the MIPS module responsible for each task. BR = boolean regression, LR = linear regression, SR = symbolic regression, and NA means MIPS is not expected to solve this problem. Green means success and red means failure.	187
7.2	AutoML architecture search results. All networks achieved 100% accuracy on at least one test batch.	217
9.1	Aggregate model accuracy on days of the week and months of the year modular arithmetic tasks. Performance broken down by problem instance in section 9.7.9.	247
9.2	Weekdays finegrained results. Row ommited if both models get it correct.	274
9.3	Months finegrained results. Row ommited if both models get it correct.	275
9.4	Highest intervention effect attention heads from fine-grained attention head patching, as well as EVR results with one hot α, β and one hot α, β, γ	278

Chapter 1

Introduction

1.1 The most interesting phenomenon in the universe

Our universe contains complexity, and order, and beauty, at many levels. At each level, from fundamental particles, to atoms, to molecules, to systems of many particles, to biological life and beyond, nature follows physical laws. And with great effort, we can discover these laws. If we look closely enough, and think carefully enough, we can create explanations with wide reach, explanations that organize the apparent complexity of the world under a simpler set of principles.

Much of the complexity that we encounter in our everyday lives is the complexity of the biological world. We are surrounded by organisms with intricate structure, structure which, we now know, was produced by evolution over very long timescales. But there is another level of complexity to the world above this—a universe of things produced not over evolutionary time, but rather as the result of what organisms **learn** during the course of their lives. Indeed, most of the complexity we experience at human scales in our environment is this sort of complexity—of ideas and artifacts which result from organisms like ourselves learning, thinking, planning, and acting to shape our environment. It is the complexity of **intelligence**.

How wonderful and strange and powerful a thing intelligence is. That we have some capacity to *learn* during our lives enables language and culture and progress, and with it radio telescopes and the splitting of the atom and, of course, the writing of PhD theses! Lately, human intelligence has been applied to creating something new, something with the potential to change our status as a species and put us in contact with other minds broadly more capable than our own for the first time. We are creating artificial systems that can learn too: artificial intelligence (AI).

Over the last decade or so, we have produced AI systems that can classify images [1], translate between languages [2], play Go [3], generate images [4], and solve graduate-level math and science problems [5] at or above the performance of human experts. Single, unified AI systems today can complete software engineering tasks that take humans hours [6], diagnose complex medical cases [7], and arguably pass the Turing Test [8,9]. This progress has come from the training of neural networks, with methods collectively called *deep learning* [10].

With these new forms of intelligence in the world, our ability to study intelligent systems has fundamentally changed. In contrast to biological brains, where experiments are slow and difficult, with artificial neural networks we can perform arbitrary experiments, making arbitrary interventions on their internals, with the ability to observe their full state as they operate and learn.

There has never been a better time to study how minds work. This thesis is about understanding these new, artificial minds in our world—understanding the things they learn, and how they come to learn them.

1.2 A brief history of deep learning

Neural networks are a class of parameterized functions $f_{\theta}(\mathbf{x})$ loosely inspired by the structure of the brain. As functions, they transform an input $\mathbf{x} \in \mathbb{R}^n$ (or a sequence of vectors $\mathbf{x} \in \mathbb{R}^{s \times n}$) into an output $\mathbf{y} \in \mathbb{R}^m$ (or sequence of vectors $\mathbf{y} \in \mathbb{R}^{s \times m}$). This computation is accomplished by a series of intermediate steps called *layers*, which typically consist of a large number of simple, homogeneous computational units called *neurons* operating in parallel. The parameters θ are called the *weights* of the network. The weights determine the computation the network performs, and so by changing the weights one changes the function computed by the network. The problem of *learning* is finding weights θ such that the computation f_{θ} implemented by the network has properties we desire.

Very early discussions of neural network models and learning were often motivated more by understanding the brain than by building useful artificial systems. Some examples include McCulloch and Pitts [11], Hebb [12], and Turing [13]. Of particular note is Frank Rosenblatt’s¹ work on Perceptrons [18–20]. Perceptrons consisted of layers of computational units which could learn with a primitive “error correction procedure” (there were successful demonstrations on some toy perceptual tasks), though Minsky famously pointed out that there were basic limits to what this neural network architecture could compute [21].²

¹I once checked out a wonderful but obscure book from the MIT library called *Communication with Extraterrestrial Intelligence*, in which Carl Sagan compiles the proceedings of a 1971 conference on the Search for Extraterrestrial Intelligence. This conference was structured around the Drake Equation [15], which estimates the number of alien civilizations N that we might be able to communicate with as a product $N = R_* \cdot f_p \cdot n_e \cdot f_l \cdot f_i \cdot f_c \cdot L$, where R_* is the rate of star formation in our galaxy, f_p is the fraction of stars which have planets, n_e the average number of planets that could potentially support life orbiting stars that have planets, f_l is the fraction of those planets which eventually give rise to any sort of life, f_i is the fraction of planets with life that eventually evolve “intelligent” life, f_c is the fraction of planets with intelligent life that develop communications technology that we could detect and L is the length of time those civilizations emit such signals (potentially the duration of the civilization). At this conference, there were presentations and discussions about each parameter of the Drake Equation from astronomers, biologists, psychologists, etc., including Freeman Dyson, Francis Crick, Marvin Minsky, and many others. In one discussion about f_p and n_e , about the problem of detecting exoplanets, Carl Sagan mentioned that a colleague of his at Cornell, a Frank Rosenblatt, had proposed a method to detect exoplanets [16]. When I read this, I thought “surely this is not the same Frank Rosenblatt who worked on Perceptrons!”, but it turns out that they are indeed the same person, to my delight. Frank had become interested in the SETI question in the final years of his life (he died tragically in a boating accident in 1971), and had shifted his research towards it despite still being up for tenure (!) and against the advice of his colleagues [17]. I bring this up to make a personal connection, which is that my first job in science was in the SETI field, working on the Breakthrough Listen initiative. The common thread between SETI and neural networks—for me, and probably for Frank too—is an interest in the nature and origin of intelligence. Today, the connection between these fields is not just via the f_i parameter in the Drake equation however, but also through the duration of technological civilization L , given the potential risks posed by advanced forms of this technology.

²Some argue that *Perceptrons* [21] caused a multi-decade delay in AI progress by steering researchers away from neural networks, for reasons that ended up being a non-sequitur: the expressivity limitations pointed out by Minsky are trivially overcome with modern architectures. I once heard someone suggest that this narrative of the history of AI in the 20th Century is consistent with an “anthropic shadow” [22]. The idea here is that if AI systems are highly existentially dangerous, the only worlds like ours in which anyone would be reflecting on the history of the AI field

Over time, a variety of neural network architectures and learning algorithms emerged that made neural networks sufficiently expressive and trainable to handle challenging real-world tasks. The simplest modern neural network architecture is the “multi-layer perceptron” (MLP)³. The layers of an MLP are simply affine transformations followed by an element-wise nonlinearity $\mathbf{x} \mapsto \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ where element-wise means that a function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is applied to each coordinate: $\sigma(\mathbf{z})_i = \sigma(z_i)$. If $\mathbf{x} \in \mathbb{R}^n$, then $\mathbf{W} \in \mathbb{R}^{m \times n}$ where m is the “width” of the layer. m is also the number of artificial “neurons” in the layer. The layer $\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ can be thought of as m parallel neurons where neuron i computes $\sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$ where $\mathbf{w}_i$ is the i th row of $\mathbf{W}$ ⁴. Layers are composed together to give the network “depth”. For instance, a two-layer network (also called a “single hidden layer” network) has a form like:

$$\mathbf{x} \mapsto \mathbf{W}_2 \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2.$$

It can be shown that neural networks as simple as this can approximate any continuous function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ (defined on a compact subset of $\mathbb{R}^n$) with arbitrary precision if the number of neurons in the hidden layer (the number of rows of $\mathbf{W}_1$) can be made arbitrarily large [23,24]. However, some functions cannot be *efficiently* approximated with a two-layer network [25], and it is often essential to make networks *deep*—composed of a large number of layers. To increase the size of our networks, we can increase both their depth (the number of layers) and the width (the number of neurons per layer).

Depending on the task, we may or may not apply a nonlinearity at the last network layer. If we want to turn the network’s output into a probability distribution, we can apply the softmax:

$$S(y)_i = \frac{\exp(y_i/T)}{\sum_j \exp(y_j/T)}$$

where T is a “temperature” parameter that adjusts the sharpness of the output distribution.⁵

A very large number of different neural network architectures have been explored. But what most of them have in common is that they consist of a large number of layers, where the output of one layer is the input of another layer, and where layers consist of simple neuron-like computational units operating in parallel.

Neural networks are *trained*. Their parameters are initially set randomly, and are then updated many times via an optimization procedure that aims to improve some measure of performance of the function f_θ implemented by the network. This optimization is performed with *gradient descent*. Given some “loss function” L that assigns a real-valued score to a neural network f_θ ,

are ones in which AI progress was delayed. Since AI progress was delayed, perhaps we then ought to update towards AI being existentially dangerous. Fortunately, there are many issues with this argument. For one, Minsky’s book may not have delayed AI progress by much in the first place, since an essential bottleneck to AI progress has simply been the availability of compute, which has grown consistently over decades.

³The name “multi-layer perceptron” is somewhat of a misnomer since these networks do not suffer from the limitations of Rosenblatt’s Perceptrons.

⁴If σ is a step function, the connection to biological neurons becomes plausible, since the neuron “fires” if a weighted combination of its input signals exceeds a threshold. Many modern choices of “activation function” σ can be loosely interpreted as “firing” when their input exceeds some threshold, for instance $\sigma(z) = \text{ReLU}(z) = \max(0, z)$ “fires” when $z > 0$.

⁵Note the connection to the Boltzmann distribution $p_i \propto \exp(-E_i/T)$.

gradient descent requires a method to compute $\nabla_{\theta} L(f_{\theta})$. This is solved by “backpropagation” [26]. Gradient descent updates the parameters of the network so as to locally decrease the loss:

$$\theta^{t+1} = \theta^t - \eta \nabla_{\theta} L(f_{\theta^t}).$$

This process is referred to as “training” and each update is called a “step” and η is called the “learning rate”. Each step only decreases the loss by a small amount, but after a large number of steps networks often learn solutions which approximately minimize the loss. A common pattern is for our loss functions to measure the network’s prediction error on a population of samples. For instance, if we wanted our network to perform “classification” on some collection of inputs, $\{(\mathbf{x}_i, l_i)\}_{i=1}^n$, we could use the network’s mean cross-entropy loss on that dataset:

$$L(f_{\theta}) = -\frac{1}{n} \sum_{i=1}^n \log(S(f_{\theta}(\mathbf{x}_i))_{l_i})$$

where $S(f_{\theta}(\mathbf{x}_i))$ is the network’s output on sample $\mathbf{x}_i$ after applying the softmax to the outputs of the network. This loss is minimized when the network predicts the true label l_i with probability 1 for each sample i in the dataset. Computing this loss requires running the network on all samples $f_{\theta}(\mathbf{x}_i)$ of the dataset, so to reduce the computational cost, the loss is typically computed on a random subset of dataset samples called a “mini-batch” during each step of training, giving a noisy approximation of the loss on the whole dataset. By increasing the size of the training dataset, we can perform more steps of training without repeating data, or we can increase the size of each mini-batch, giving us a better approximation of the true loss on each step.

This was a highly simplified overview of the ideas of deep learning. However, this basic approach has proven remarkably successful when applied at *scale*. It has been found repeatedly that when we scale up the size of our networks and the amount of data they are trained on that networks learn solutions that perform well and generalize in impressive ways. Indeed, much of the progress in deep learning over the last several years has come simply from scaling up the compute used to train neural networks—training larger networks on more data [27]. Interestingly, there appears to be a kind of predictable order to the improvements that come from scale. Many works have observed *scaling laws*, where the loss scales as a power law in the number of parameters in the network $L \propto N^{-\alpha_N}$ and the number of samples in the training dataset $L \propto D^{-\alpha_D}$ [28–31]. These laws played an important role in motivating OpenAI to scale up language models, which led to the explosion of interest and investment in AI since late 2022.

1.3 Deep learning at the microscale and macroscale

Neural networks are understood at two levels: the *microscale* and the *macroscale*.

By “microscale”, I mean that we have a perfect low-level understanding of the state and dynamics of our networks. We implement gradient descent ourselves, and so we understand the “rules” by which the weights of our networks change at each step, and we have full access to the internal state of our systems at all times. There is no mystery to the low-level “physics” of neural networks.

By “macroscale”, I mean that we have an empirical understanding of how certain high-level variables of our systems depend on each other in practice. For instance, with scaling laws, we

know that the network’s loss changes predictably with the network’s size (its number of weights). We can study empirically how a variety of quantities like this vary with each other. This science is analogous to thermodynamics in physics.

However, both the microscale and the macroscale descriptions treat neural networks as a black box. While we understand how each network weight changes at each step of training, this knowledge isn’t useful for predicting any meaningful properties of the network as a whole. Knowing the value of any individual network parameter, or even every network parameter, doesn’t immediately give you insight into why the network behaves as it does on any input (without simply running that input on the whole network). It also doesn’t immediately explain the macroscale physics of these systems. Indeed, there is a similarity here to the history of the thermodynamic relations in physics, since in both fields the high-level regularities were empirically documented before the theoretical understanding of why they arise was found. When studying the macroscale relationships in deep learning, it is common for authors to explicitly acknowledge that they are doing a purely empirical science, treating networks as a black box without attempting to understand *why* these macroscale regularities arise [32].

It is natural for us to wonder whether there could be some other approach to understanding these systems, a level of description which would bridge the microscale and macroscale and give us good explanations of *why* neural networks behave and scale as they do.

1.4 A “cognitive revolution” in the science of deep learning

Over the last few years, a different kind of science of deep neural networks has emerged. This science is aimed at understanding the internal *mechanisms* that neural networks learn, and how and why these mechanisms are learned over training.

This line of work, called “interpretability” or “mechanistic interpretability” [33], seeks to understand the internal representations [34–37] and computations [38–41] that give rise to neural network behavior. This work is bold. It assumes, contrary to the prevailing view of many researchers in the broader field, that neural networks need not be black boxes, and that their computation admits a description that we can understand.

The growth of this subfield parallels the “cognitive revolution” in psychology—the development of cognitive science in the middle of the 20th century against the prevailing paradigm of behaviorism. Mechanistic interpretability aims to be a kind of cognitive science for deep learning against the “black box” behaviorist thinking of many deep learning researchers today.

1.5 The mesoscale: a physics of the “parts”

1.5.1 Mechanisms and their properties

Interpretability studies the “mechanisms” that neural networks learn. I will not try to give a formal definition of a mechanism here—unfortunately, the field does not have one. Instead, we have a large number of case studies where researchers have found, through great effort, that neural networks implement specific computations in intelligible ways. In these case studies, we find that neural networks represent specific computational variables via particular patterns of

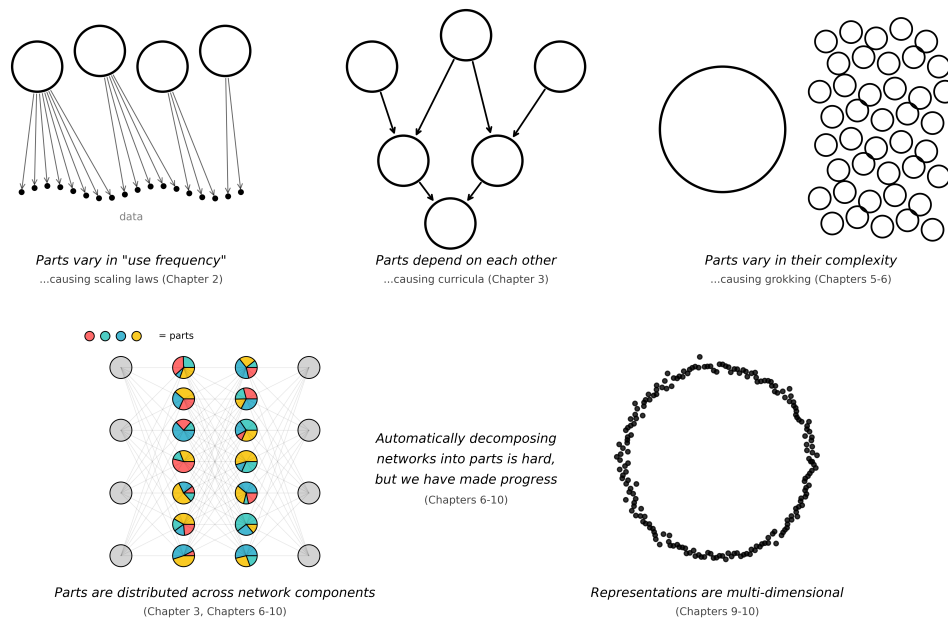


Figure 1.1: We suggest that the computation that neural networks perform can be decomposed into “parts” (mechanisms / modules). The chapters of this thesis address questions about the properties of these parts, and how we can automatically find decompositions of neural networks into parts.

activity in their hidden layers. We find that these representations often have striking geometric structure. And we find that these representations are acted on by the network’s layers to form other representations in later layers, so that altogether the layers of the network implement intelligible algorithms to solve specific prediction problems. These mechanisms tend to be quite narrow—on unrelated network inputs, a different set of mechanisms are involved in the network’s operation.

I suggest that the study of these mechanisms provides a “mesoscale” for understanding deep learning. Mechanisms lie between the microscale and the macroscale. They are not individual network weights, but are the result of a large number of network weights taking particular values to enable some computation. They are also not the network as a whole. The network implements a very large number of different mechanisms—computational “parts” that are needed at different times. We can study how individual mechanisms form during training, bridging the microscale dynamics with the “parts” at the mesoscale [38,40,42]. And we can understand the macroscopic properties of the network as in some way relating to the properties of these parts, bridging the mesoscale with the macroscale [43].

There is a sentiment in interpretability that the field is more akin to biology than physics [44]. This view is sensible, since the mechanisms that neural networks learn are diverse, the result of a learning process interacting with complex and messy data from the real world. However, I expect that if we look closely enough, there will be a deep regularity to these mechanisms. For instance, across networks trained on similar data, mechanisms may be learned universally and in the same order during training [43,45]. There may also be a regularity to the statistics of how frequently

mechanisms are used and how complex they are [43].

This thesis studies neural network mechanisms—the “parts” that make up neural network computation. We explore questions like the following, summarized in Figure 1.1:

- If we can decompose networks into modular parts, what might neural scaling laws tell us about the statistics of these parts?
- How do these parts interact with each other? Do some parts need to be learned before others can be learned?
- When different parts can both predict the training data, but are learned at different rates, require different amounts of network capacity, and generalize in different ways, which parts will “win”, and how will the network generalize over training?
- How can we decompose neural network computation into parts in the first place? Is there a “right” unit of decomposition?
- Can we hope to enumerate all the parts in the network, or will this be intractable?

1.5.2 What does this say about the world?

Richard Feynman began his “Lectures on Physics” [46] with the following thought experiment:

If, in some cataclysm, all of scientific knowledge were to be destroyed, and only one sentence passed on to the next generations of creatures, what statement would contain the most information in the fewest words? I believe it is the atomic hypothesis (or the atomic fact, or whatever you wish to call it) that all things are made of atoms—little particles that move around in perpetual motion, attracting each other when they are a little distance apart, but repelling upon being squeezed into one another. In that one sentence, you will see, there is an enormous amount of information about the world, if just a little imagination and thinking are applied.

What might a similar statement be for understanding neural networks, and more broadly minds and intelligence? I think we ought to consider a kind of atomic hypothesis for minds—that the computation of neural networks, and minds more generally, decomposes into parts, which perform specific computations, and individual neural networks can only implement so many of these parts. The statement might be something like:

Many different computations occur in our world in the processes that produce data, and in order for neural networks to predict the world, they must learn to implement these computations. Finite neural networks have finite capacity, and can implement only a limited number of computations. During learning, the computations that explain the most data using the least capacity eventually “win”, and so are “replicated” within the network.⁶

⁶Cf. Dawkins [47] and Minsky [48].

1.6 Contents and Contributions

Following this introduction, this thesis contains 9 chapters, each a paper from my PhD. I am first-author on 4 of these, co-first author on 2, and a middle author on 3. The thesis is roughly split into two parts. Chapters 2–6 discuss neural network training dynamics and scaling laws through the perspective of modularity/mechanisms, and chapters 7–10 address the challenge of decomposing already-trained neural networks into mechanisms. My contributions to each paper are as follows:

Chapter 2: Quantization Model of Neural Scaling

Summary: This paper articulates the “quanta hypothesis” that neural networks decompose into a large number of computational modules that are learned discretely. It offers a model of neural scaling laws based on the hypothesis that the frequency at which these computational modules are needed by the model naturally follow a power law.

Contributions: First author. I conceived the core idea of the paper and the multitask sparse parity task. I ran most experiments (all main text), made most figures (all main text), and led the drafting and revising of the manuscript. My coauthors ran experiments and made figures for the appendix, contributed multiple ideas for experiments throughout the paper and ideas for how to present results and design the paper, and helped draft and revise the manuscript.

Eric J. Michaud, Ziming Liu, Uzay Girit, and Max Tegmark. “The quantization model of neural scaling.” *Advances in Neural Information Processing Systems* 36 (2023): 28699–28722.

Chapter 3: On the Creation of Narrow AI: Hierarchy and Nonlocality of Neural Network Skills

Summary: This paper studies the possibility that computational modules may depend on each other hierarchically, where some must be learned before others can be learned. It also explores how the circuitry for specific skills can be distributed across the network, despite being ultimately simple.

Contributions: First author. Max and I came up with the compositional multitask sparse parity task (CMSP). I ran the experiments on CMSP and the language model experiments. I drafted most of the manuscript, with help on revisions from Asher and Max.

Eric J. Michaud, Asher Parker-Sartori, and Max Tegmark. “On the creation of narrow AI: hierarchy and nonlocality of neural network skills.” *Advances in Neural Information Processing Systems* 39 (to appear), 2025. Preprint: [arXiv:2505.15811](https://arxiv.org/abs/2505.15811).

Chapter 4: Precision Machine Learning

Summary: Among other things, this paper considers how modularity can change neural scaling laws, when neural networks are used as approximators of continuous functions.

Contributions: First author. I ran all experiments, made all figures, and drafted the manuscript. The project direction was set by Max, with many ideas and feedback coming from meetings with Ziming and Max.

Eric J. Michaud, Ziming Liu, and Max Tegmark. “Precision machine learning.” *Entropy* 25, no. 1 (2023): 175.

Chapter 5: Towards Understanding Grokking: An Effective Theory of Representation Learning

Summary: This paper studies how structured representations form when neural networks “grok”—generalize long after initially overfitting their training data.

Contributions: Middle author. I ran MNIST experiments and helped draft the manuscript.

Ziming Liu, Ouail Kitouni, Niklas S. Nolte, Eric J. Michaud, Max Tegmark, and Mike Williams. “Towards understanding grokking: An effective theory of representation learning.” *Advances in Neural Information Processing Systems* 35 (2022): 34651–34663.

Chapter 6: Omnigrok: Grokking Beyond Algorithmic Data

Summary: This paper studies how the phenomenon of grokking relates to the weight norm of neural networks.

Contributions: Middle author. I ran some of the MNIST experiments and also conceived of and ran the experiment showing that constraining weight norm accelerates grokking on modular addition.

Ziming Liu, Eric J. Michaud, and Max Tegmark. “Omnigrok: Grokking beyond algorithmic data.” *International Conference on Learning Representations* (2023).

Chapter 7: Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code

Summary: This paper shows, extremely directly, that neural networks are not black boxes. We show that we can translate neural networks directly into Python programs, which can be run instead of the network.

Contributions: Co-first author. I led the creation of the tasks/datasets used in the paper, and devised and ran the architecture search algorithm used to train minimum-sized networks on each task. I also helped draft and revise the manuscript.

Eric J. Michaud, Isaac Liao, Vedang Lad, Ziming Liu, Anish Mudide, Chloe Loughridge, Zifan Carl Guo, Tara Rezaei Kheirkhah, Mateja Vukelić, and Max Tegmark. “Opening the AI black box: Distilling machine-learned algorithms into code.” *Entropy* 26, no. 12 (2024): 1046.

Chapter 8: The Geometry of Concepts: Sparse Autoencoder Feature Structure

Summary: This paper studies the structure of the “feature” directions learned by sparse autoencoders for large language models.

Contributions: Co-first author. I ran almost all experiments for the “Brain-scale” section, wrote that same section, and helped to write and revise the whole paper, especially the introduction.

Yuxiao Li, Eric J. Michaud, David D. Baek, Joshua Engels, Xiaoqing Sun, and Max Tegmark. “The geometry of concepts: Sparse autoencoder feature structure.” *Entropy* 27, no. 4 (2025): 344.

Chapter 9: Not All Language Model Features Are One-Dimensionally Linear

Summary: This paper finds evidence of multi-dimensional “feature manifolds” in neural network representations.

Contributions: Middle author. I ran the experiments for the automated discovery of multi-dimensional features in GPT-2, and the experiments training SAEs on toy feature manifolds. I helped draft and revise the manuscript.

Joshua Engels, Eric J. Michaud, Isaac Liao, Wes Gurnee, and Max Tegmark. “Not all language model features are one-dimensionally linear.” *International Conference on Learning Representations* (2025).

Chapter 10: Understanding Sparse Autoencoder Scaling Laws in the Presence of Feature Manifolds

Summary: This paper develops a model of scaling laws for sparse autoencoders, in the particular case where there are feature manifolds in the activations.

Contributions: First author. I performed the mathematical analysis, ran almost all experiments, and drafted the manuscript, with feedback from Liv and Tom. Liv conceived of the core idea that feature manifolds could cause pathological SAE scaling.

Eric J. Michaud, Liv Gorton, and Tom McGrath. “Understanding sparse autoencoder scaling in the presence of feature manifolds.” *Mechanistic Interpretability Workshop at NeurIPS* (2025). Preprint: [arXiv:2509.02565](https://arxiv.org/abs/2509.02565).

References

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks.” *Advances in neural information processing systems*, 25, 2012.
- [2] I. Sutskever, O. Vinyals, and Q. V. Le. “Sequence to sequence learning with neural networks.” *Advances in neural information processing systems*, 27, 2014.
- [3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search.” *nature*, 529(7587), 2016, pp. 484–489.
- [4] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. “Zero-shot text-to-image generation.” In: *International conference on machine learning*. Pmlr, 2021, pp. 8821–8831.
- [5] D. Rein, B. L. Hou, A. C. Stickland, J. Petty, R. Y. Pang, J. Dirani, J. Michael, and S. R. Bowman. “Gpqa: A graduate-level google-proof q&a benchmark.” In: *First Conference on Language Modeling*. 2024.
- [6] T. Kwa, B. West, J. Becker, A. Deng, K. Garcia, M. Hasin, S. Jawhar, M. Kinniment, N. Rush, S. Von Arx, et al. “Measuring ai ability to complete long tasks.” *arXiv preprint arXiv:2503.14499*, 2025.
- [7] H. Nori, M. Daswani, C. Kelly, S. Lundberg, M. T. Ribeiro, M. Wilson, X. Liu, V. Sounderajah, J. Carlson, M. P. Lungren, et al. “Sequential Diagnosis with Language Models.” *arXiv preprint arXiv:2506.22405*, 2025.
- [8] A. M. Turing. “Computing machinery and intelligence.” In: *Parsing the Turing test: Philosophical and methodological issues in the quest for the thinking computer*. Springer, 2007, pp. 23–65.
- [9] C. R. Jones and B. K. Bergen. “Large language models pass the turing test.” *arXiv preprint arXiv:2503.23674*, 2025.
- [10] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning.” *nature*, 521(7553), 2015, pp. 436–444.
- [11] W. S. McCulloch and W. Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity.” *The Bulletin of Mathematical Biophysics*, 5, 1943, pp. 115–133. DOI: [10.1007/BF02478259](https://doi.org/10.1007/BF02478259).
- [12] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. New York: John Wiley & Sons, 1949.
- [13] A. M. Turing. *Intelligent Machinery*. Tech. rep. Unpublished NPL report; reprinted in D. C. Ince (ed.), *Mechanical Intelligence: Collected Works of A. M. Turing*, North-Holland, 1992. Teddington, UK: National Physical Laboratory, 1948.
- [14] C. Sagan, ed. *Communication with Extraterrestrial Intelligence*. English-language proceedings of the First Soviet–American Conference on Communication with Extraterrestrial Intelligence, Byurakan Astrophysical Observatory, Armenia, 5–11 Sept. 1971. Cambridge, MA: The MIT Press, 1973. 428 pp. ISBN: 9780262191067.
- [15] F. D. Drake. “The radio search for intelligent extraterrestrial life.” *Current aspects of exobiology*, 1965, pp. 323–345.

- [16] F. Rosenblatt. “A Two-Color Photometric Method for Detection of Extra-Solar Planetary Systems.” *Icarus*, 14(1), Feb. 1971, pp. 71–93. DOI: [10.1016/0019-1035\(71\)90103-5](https://doi.org/10.1016/0019-1035(71)90103-5).
- [17] H. L. Carey. “Tribute to Dr. Frank Rosenblatt.” *Congressional Record: Proceedings and Debates of the 92d Congress, First Session*, 1971. Archived from the original (PDF) on 26 February 2014. Retrieved 24 December 2021, pp. 1–7.
- [18] F. Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton*. Tech. rep. Report 85-460-1. Project PARA. Buffalo, NY: Cornell Aeronautical Laboratory, Jan. 1957.
- [19] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.” *Psychological Review*, 65(6), 1958, pp. 386–408. DOI: [10.1037/h0042519](https://doi.org/10.1037/h0042519).
- [20] F. Rosenblatt et al. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Vol. 55. Spartan books Washington, DC, 1962.
- [21] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA: MIT Press, 1969. ISBN: 0262130432.
- [22] M. M. Ćirković, A. Sandberg, and N. Bostrom. “Anthropic shadow: observation selection effects and human extinction risks.” *Risk Analysis: An International Journal*, 30(10), 2010, pp. 1495–1506.
- [23] G. Cybenko. “Approximation by Superpositions of a Sigmoidal Function.” *Mathematics of Control, Signals, and Systems*, 2(4), 1989, pp. 303–314. DOI: [10.1007/BF02551274](https://doi.org/10.1007/BF02551274).
- [24] K. Hornik, M. Stinchcombe, and H. White. “Multilayer Feedforward Networks Are Universal Approximators.” *Neural Networks*, 2(5), 1989, pp. 359–366. DOI: [10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8).
- [25] H. W. Lin, M. Tegmark, and D. Rolnick. “Why does deep and cheap learning work so well?” *Journal of Statistical Physics*, 168(6), 2017, pp. 1223–1247.
- [26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors.” *Nature*, 323(6088), 1986, pp. 533–536. DOI: [10.1038/323533a0](https://doi.org/10.1038/323533a0).
- [27] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. “Compute trends across three eras of machine learning.” In: *2022 international joint conference on neural networks (IJCNN)*. IEEE. 2022, pp. 1–8.
- [28] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. “Deep learning scaling is predictable, empirically.” *arXiv preprint arXiv:1712.00409*, 2017.
- [29] J. S. Rosenfeld, A. Rosenfeld, Y. Belinkov, and N. Shavit. “A constructive prediction of the generalization error across scales.” *arXiv preprint arXiv:1909.12673*, 2019.
- [30] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling laws for neural language models.” *arXiv preprint arXiv:2001.08361*, 2020.
- [31] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T. B. Brown, P. Dhariwal, S. Gray, et al. “Scaling laws for autoregressive generative modeling.” *arXiv preprint arXiv:2010.14701*, 2020.

- [32] P. Nakkiran. “Towards an Empirical Theory of Deep Learning.” Dissertation presented to The School of Engineering and Applied Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science. PhD thesis. Cambridge, Massachusetts: Harvard University, July 2021.
- [33] N. Saphra and S. Wiegrefe. “Mechanistic?” *arXiv preprint arXiv:2410.09087*, 2024.
- [34] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. “Emergent world representations: Exploring a sequence model trained on a synthetic task.” *ICLR*, 2023.
- [35] S. Marks and M. Tegmark. “The geometry of truth: Emergent linear structure in large language model representations of true/false datasets.” *arXiv preprint arXiv:2310.06824*, 2023.
- [36] A. Templeton et al. “Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet.” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html>.
- [37] J. Engels, E. J. Michaud, I. Liao, W. Gurnee, and M. Tegmark. “Not All Language Model Features Are One-Dimensionally Linear.” In: *The Thirteenth International Conference on Learning Representations (ICLR)*. Poster. Singapore: OpenReview.net, Apr. 2025. URL: <https://openreview.net/forum?id=d63a4AM4hb>.
- [38] C. Olsson et al. “In-context Learning and Induction Heads.” *Transformer Circuits Thread*, 2022. URL: <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [39] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt. “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small.” *arXiv preprint arXiv:2211.00593*, 2022.
- [40] N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.
- [41] J. Lindsey et al. “On the Biology of a Large Language Model.” *Transformer Circuits Thread*, 2025. URL: <https://transformer-circuits.pub/2025/attribution-graphs/biology.html>.
- [42] G. Reddy. “The mechanistic basis of data dependence and abrupt learning in an in-context classification task.” *arXiv preprint arXiv:2312.03002*, 2023.
- [43] E. Michaud, Z. Liu, U. Girit, and M. Tegmark. “The quantization model of neural scaling.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 28699–28722.
- [44] C. Olah. *The elegance of ML is the elegance of biology, not the elegance of math or physics. Simple gradient descent creates mind-boggling structure and behavior, just as evolution creates the awe inspiring complexity of nature.* [Tweet]. Twitter/X. Accessed: 2025-09-14. June 2022. URL: <https://x.com/ch402/status/1533164918886703104>.
- [45] C. Tigges, M. Hanna, Q. Yu, and S. Biderman. “LLM circuit analyses are consistent across training and scale.” *Advances in Neural Information Processing Systems*, 37, 2024, pp. 40699–40731.
- [46] R. P. Feynman. “The Feynman lectures on physics.” (*No Title*), 1, 1963, p. 46.
- [47] R. Dawkins. *The Selfish Gene*. 1st. Oxford, UK: Oxford University Press, 1976.
- [48] M. Minsky. *Society of mind*. Simon and Schuster, 1986.

Chapter 2

The Quantization Model of Neural Scaling^{*}

This paper articulates the core idea of this thesis: that the computation that neural networks perform is decomposable into a set of simpler parts. While networks as a whole might be complex, their computation on any individual sample is simpler, because only a small subset of these mechanisms is relevant to that sample. In this work, we construct a model of neural scaling based on this idea. To observe neural scaling laws, we just need to assume that the “use frequencies” of these mechanisms naturally follow a power law.

Abstract

We propose the *Quantization Model* of neural scaling laws, explaining both the observed power law dropoff of loss with model and data size, and also the sudden emergence of new capabilities with scale. We derive this model from what we call the *Quantization Hypothesis*, where network knowledge and skills are “quantized” into discrete chunks (*quanta*). We show that when quanta are learned in order of decreasing use frequency, then a power law in use frequencies explains observed power law scaling of loss. We validate this prediction on toy datasets, then study how scaling curves decompose for large language models. Using language model gradients, we automatically decompose model behavior into a diverse set of skills (quanta). We tentatively find that the frequency at which these quanta are used in the training distribution roughly follows a power law corresponding with the empirical scaling exponent for language models, a prediction of our theory.¹

2.1 Introduction

In the aggregate, larger neural networks trained on more data perform better than smaller neural networks trained on less data, in a predictable way. Across a range of studies, mean test loss has been observed to decrease as a power law in both the number of network parameters ($L \propto N^{-\alpha_N}$)

^{*}Eric J. Michaud, Ziming Liu, Uzay Girit, and Max Tegmark. “The quantization model of neural scaling.” *Advances in Neural Information Processing Systems* 36 (2023): 28699–28722.

¹Project code can be found at: <https://github.com/ejmichaud/quantization-model>.

and the number of training samples ($L \propto D^{-\alpha_D}$) [1–7]. Although aggregate performance changes smoothly with scale, when particular capabilities are examined, larger models often have emergent abilities, i.e., qualitatively different performance than smaller models [8,9]. Understanding and reconciling both facets of scaling – the predictable power law decrease in loss and the emergence of new capabilities at scale – is of both theoretical and practical interest [10]. Understanding how scaling changes what neural networks learn is entangled with core questions: what are deep neural networks doing internally, and will they will continue to improve with scale?

Recent studies of the internals of neural networks have found a variety of impressive algorithms learned by gradient descent [11–15]. As more work is put into understanding the structures learned by neural networks (the task of *mechanistic interpretability*), we may find more and more *circuits* [11,16] in models, intelligible internal algorithms for accomplishing prediction in specific contexts. Can such analysis be scaled up to frontier models [17]? Two assumptions which, if true, would make us more optimistic about mechanistically understanding large models include (1) decomposability/modularity/sparsity [18–21] – that large models are decomposable into parts, and only a small number of these parts are relevant to the model’s behavior on any given sample and (2) universality [11,22–24] – that similar structures recur across models of increasing size. Recently, Olsson et al. [25] found encouraging evidence for universality of “induction heads” across LLMs and found that these emerge in a discrete transition during training.

In this paper, we articulate the *Quantization Hypothesis*, a set of informal conjectures about the *decomposability* of networks into smaller parts, the *universality* of computations performed across model scales, the *discreteness* of what models learn, and about how properties of the data distribution produce power law neural scaling. In particular, we hypothesize that to many prediction problems, there corresponds a particular enumerable set of indivisible pieces of knowledge or skills that models must learn, and that model performance is determined by *which* of these elements models successfully learn. We call these basic building blocks of model performance the **quanta**:

Quantum (plural quanta): An indivisible computational module that, for example, retrieves a fact, implements an algorithm, or more generally corresponds to some basic skill possessed by a model.

We use this terminology in analogy to Max Planck’s assumption in 1900 that energy is quantized into discrete chunks (quanta) – here we imagine that knowledge/skills are quantized into discrete chunks (quanta). Since “quantization” is commonly used in machine learning in the context of low-precision arithmetic, we suggest “knowledge quantization” or “skill quantization” to refer to our notion of quantization. We will see that a Zipfian distribution governing the “use frequency” of the quanta produces power law neural scaling, where the effect of scaling is to learn an increasing number of discrete quanta, and smooth scaling laws average over small discrete jumps in model performance.

This paper is organized as follows: in Section 2.2 we give a theoretical model of power law neural scaling from the Quantization Hypothesis. In Section 2.3 we construct toy datasets satisfying the hypothesis, where smooth power laws average over many discrete jumps in model performance. In Section 2.4 we then analyze how power law scaling decomposes for real LLMs. In Section 2.5, we develop a method for automatically discovering quanta in language models by

LLM skill “quanta” auto-discovered in text													
Cluster 50: incrementing numerical sequences	Cluster 100: predicting newlines in line length limited text												
<pre> 01- Mi Querencia (Simón Díaz)\n 02- Tonada De Luna Llena (Simón Díaz)\n 03- Sabana (José Salazar/Simón Díaz)\n 04- Caballo Viejo (Simón Díaz)\n 05- Todo Este Campo Es Mío (Simón Díaz)\n 06- La Pena Del Becerrero (Simón Díaz)\n 07- </pre> * * from opening a through road or street for public use across said public park in the Park of The City of Riverton * * * (Emphasis supplied.) Appealing from that order, the city asserts (1) plaintiffs have no standing or right to maintain the action; (2) that the proposed road was in an undedicated part of the park; (3) that the proposed road was an access road and not a through street or part of the city’s street system; (4 <pre> 4. _Introduction_\n 5. Chapter 1: What Is Trust?\n 6. Chapter 2: Trust Brings Rest\n 7. Chapter 3: Who Can I Trust?\n 8. Chapter 4: The Folly of Self-Reliance\n 9. Chapter 5: Trust God and Do Good (Part 1)\n 10. Chapter 6: Trust God and Do Good (Part 2)\n 11. Chapter 7: At All Times\n 12. Chapter 8 </pre> was achieved. The chosen sites were recorded as: 0 = sound (*n* = 13); 1 = first visible sign of noncavitated lesion seen only when the tooth is dried; 2 = visible noncavitated lesion seen when wet and dry; 3 = microcavitation in enamel; 4 = noncavitated lesion extending into dentine seen as an undermining shadow; 5 = small cavitated lesion with visible dentine; less than 50% of surface; 6 <table> <tr> <td>QCBlockListMsg</td><td>= 0x0a\n</td></tr> <tr> <td>GetLatestStatusMsg</td><td>= 0x0b\n</td></tr> <tr> <td>LatestStatusMsg</td><td>= 0x0c\n</td></tr> <tr> <td>PrepareBlockHashMsg</td><td>= 0x0d\n</td></tr> <tr> <td>GetViewChangeMsg</td><td>= 0x0e\n</td></tr> <tr> <td>PingMsg</td><td>= 0x0f</td></tr> </table>	QCBlockListMsg	= 0x0a\n	GetLatestStatusMsg	= 0x0b\n	LatestStatusMsg	= 0x0c\n	PrepareBlockHashMsg	= 0x0d\n	GetViewChangeMsg	= 0x0e\n	PingMsg	= 0x0f	<pre> TO PERFORM QUADRATIC REGRESSION\n ON THE TI84 GRAPHING CALCULATOR,\n DETERMINE HOW WELL THE \n REGRESSION MODEL FITS THE DATA,\n AND THEN MAKE PREDICTIONS \n USING THE REGRESSION EQUATION.\n IN STATISTICS, \n REGRESSION ANALYSIS INCLUDES\n ANY TECHNIQUES USED FOR MODELING. \n </pre> <pre> # credump is free software: you can redistribute it and/or modify\n # it under the terms of the GNU General Public License as published by\n # the Free Software Foundation, either version 3 of the License, or\n # (at your option) any later version.\n #\n # credump is distributed in the hope that it will be useful.\n </pre> <pre> <!--\n /**\n * Copyright (c) 2019, The Android Open Source Project\n */\n * Licensed under the Apache License, Version 2.0 (the "License");\n * you may not use this file except in compliance with the License.\n </pre> <pre> *\n Pursuant to 5TH CIR. R. 47.5, the court has determined\n that this opinion should not be published and is not precedent\n except under the limited circumstances set forth in 5TH CIR. \n </pre> children have a lack of maturity and an underdeveloped\n sense of responsibility, leading to recklessness, impul-\n sivity, and heedless risk-taking.... Second, children\n are more vulnerable... to negative influences and\n outside pressures, including from their family and\n peers; they have limited contro[1] over their own envi-\n
QCBlockListMsg	= 0x0a\n												
GetLatestStatusMsg	= 0x0b\n												
LatestStatusMsg	= 0x0c\n												
PrepareBlockHashMsg	= 0x0d\n												
GetViewChangeMsg	= 0x0e\n												
PingMsg	= 0x0f												

Figure 2.1: We auto-discover *quanta* – basic units of model knowledge/skill – for a language model. Here we show collections of next-token prediction samples which our method clustered together, each corresponding to some coherent model behavior. We indicate the token which was predicted from the context before it with a red highlight. We indicate newlines using “\n”. See Section 2.5 for explanation.

clustering their behavior into basic coherent skills, and analyze the statistics of these clusters, concluding in Section 2.7.

2.2 Theory

Consider the task of modeling the distribution of text on the internet. Successful prediction requires an immense amount of knowledge, and the ability to perform diverse computations, due to the immense complexity and diversity of the world and therefore of human language. For instance, in order to predict what word will come next in a conversation between two physicists, one must “know” much about physics. In order to continue the text “2534 + 7261 = ”, one must be able to perform arithmetic (for large enough numbers, memorization becomes a highly inefficient strategy) [26]. A great many distinct types of computations are present in the world in the processes that *produce* text, and so *predicting* text requires those computations to be present in our models.

In this paper, we conjecture the Quantization (or Quanta) Hypothesis:

- QH1 Many natural prediction problems decompose into an enumerable set of computations, pieces of knowledge, or skills, which models must learn to reduce loss. We call these **quanta**, and model them as being *discrete*, – they are either learned or not learned. Model performance is determined by *which* quanta have been learned.
- QH2 Some quanta are more useful for reducing loss than others, leading to a natural ordering of the quanta. We call the ordered quanta the **Q Sequence**. Optimally trained networks should therefore learn the quanta in that order. The effect of scaling is to learn *more* of the quanta in the Q Sequence, so scaling performance is simply determined by *how many* quanta are successfully learned.
- QH3 The frequencies at which the quanta are used for prediction follow a power law.

Together these can result in power law neural scaling.

We model the Quantization Hypothesis as follows, referring to the below as the “Quantization (or Quanta) Model”. Let $\mathbf{q}$ denote a bit string whose k^{th} bit $\mathbf{q}_k = 1$ if the k^{th} quantum in the Q Sequence has been learned, and $\mathbf{q}_k = 0$ otherwise. QH1 implies that the mean loss L is simply a function of $\mathbf{q}$. QH2 implies that when $n \equiv \sum_k \mathbf{q}_k$ quanta have been learned, we have $\mathbf{q}_k = 1$ for $k \leq n$. Let L_n denote the mean loss in this case. From QH3, we have that the k^{th} quantum benefits prediction on a randomly chosen sample with probability

$$p_k = \frac{1}{\zeta(\alpha + 1)} k^{-(\alpha+1)} \propto k^{-(\alpha+1)} \quad (2.1)$$

for a Zipf power law $\alpha > 0$, where $\zeta(s) \equiv \sum_{k=1}^{\infty} k^{-s}$. Let us also assume that learning the k^{th} quantum reduces average loss from b_k before it is learned to a_k after it is learned on the samples where it is utilized. If a_k and b_k are k -independent ($a_k = a$, $b_k = b$), then a model that has learned the first n quanta will have expected loss

$$\begin{aligned} L_n &= \sum_{k=1}^n a p_k + \sum_{k=n+1}^{\infty} b p_k = \sum_{k=1}^{\infty} a p_k + \sum_{k=n+1}^{\infty} (b - a) p_k \\ &\approx a + \frac{b - a}{\zeta(\alpha + 1)} \int_n^{\infty} k^{-(\alpha+1)} dk = a + \frac{b - a}{\alpha \zeta(\alpha + 1)} n^{-\alpha}. \end{aligned} \quad (2.2)$$

In other words, $L_\infty = a$ and $(L_n - L_\infty) \propto n^{-\alpha}$ is a power law.

In Section 2.8.1, we provide analogous derivations for other assumptions for a_k and b_k , and find that a range of assumptions produce curves that are exact or approximate power laws – the latter include a small logarithmic correction.

In the derivation above, we assumed that all samples are what we will refer to as *monogenic*, meaning that prediction relies on at most a single quantum, akin to how monogenic traits in biology (e.g. cystic fibrosis) depend on a single gene. By assuming that all samples are monogenic, we can write the expected loss as a sum over quanta, weighted by the fraction of samples which rely on that quanta p_k . We further explore the idea of monogenic vs. polygenic samples in Section 2.4.2. So far we have seen how the Quantization Hypothesis can produce power law scaling as a function of the number of quanta learned n . We will now give one possible mechanism by which this can translate into power law scaling in parameters, data, etc.:

Parameter scaling: In networks of finite size, network capacity can bottleneck how many quanta are learned. If we assume that all quanta require the same capacity of C network parameters, then a network with N parameters can learn roughly $n \approx N/C$ quanta. Therefore $L(N) - L_\infty \propto n^{-\alpha} \approx (N/C)^{-\alpha} \propto N^{-\alpha}$, so we get power law scaling in N with exponent $\alpha_N = \alpha$.

Data scaling (multi-epoch): For data scaling, we assume that for each quantum, a threshold of τ examples utilizing quantum k are needed in the training set for quantum k to be learned². With D training samples, approximately Dp_k samples relying on quantum k will be present, and solving for $Dp_n = \tau$ we get the last quantum to be learned will be $n \propto (D/\tau)^{1/(\alpha+1)}$ since $p_k \propto k^{-(\alpha+1)}$. Under this model, we get scaling in data samples $L(D) - L_\infty \propto n^{-\alpha} \propto (D/\tau)^{-\alpha/(\alpha+1)} \propto D^{-\alpha/(\alpha+1)}$, and so $\alpha_D = \alpha/(\alpha + 1)$. From our earlier result that $\alpha_N = \alpha$, we would therefore predict that $\alpha_D = \alpha_N/(\alpha_N + 1)$. We discuss whether this relationship holds empirically for data and parameter scaling exponents observed across a variety of studies in Section 2.8.6.

Data scaling (single-epoch): In multi-epoch training, the information contained in the training dataset can bottleneck which quanta are learned. However, the rate of convergence of SGD can also bottleneck performance. For single-epoch training, a greater number of training samples allows one to train for longer. In our model, the amount that each quantum reduces mean loss by follows a power law. If the magnitude of the gradients for learning these quanta also follow a power law, then the convergence time for each quanta may follow a power law too. If the number of steps to learn quantum k is $\propto 1/p_k$, then if the first quantum requires T steps to be learned, quantum n will require $Tn^{\alpha+1}$ steps, and so $n = (S/T)^{1/(\alpha+1)}$ quanta can be learned in S steps. This gives scaling in training steps $L(S) - L_\infty \propto n^{-\alpha} \approx (S/T)^{-\alpha/(\alpha+1)} \propto S^{-\alpha/(\alpha+1)}$, and so $\alpha_S = \alpha/(\alpha + 1)$. Under this model, multi-epoch and single-epoch data scaling exponents coincide: $\alpha_D = \alpha_S$.

2.3 Proof of concept: a toy dataset

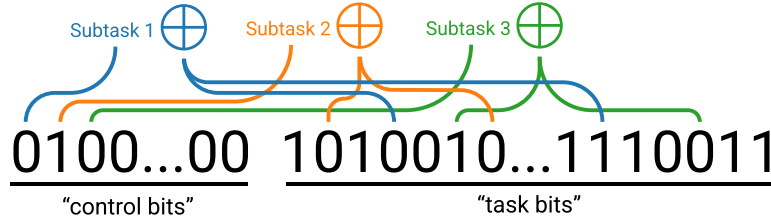
In this section, we will describe a toy dataset consisting of distinct subtasks which are power law distributed in frequency. We observe power law neural scaling in data and parameters on this task, and find that the mechanism of neural scaling coincides with our theory from Section 2.2. It is therefore possible for scaling laws to arise from the Quantization Model for data with the right structure. We leave a study of whether natural datasets (e.g. natural modeling) possess such

²This type of threshold has precedent, e.g. for the algorithmic tasks where “grokking” occurs [27].

structure to Section 2.4.

2.3.1 The “multitask sparse parity” dataset

The toy task we will construct consists of many subtasks – distinct types of inputs which each require corresponding distinct computations (quanta). For each subtask, we choose a variant of the “sparse parity” problem, recently studied in [28]. The sparse parity prediction problem is simple: given a bit string of length n , compute the parity (sum mod 2) of a fixed subset of k of those bits. We introduce an extension of this task, which we call “multitask sparse parity”. Beyond n and k , multitask sparse parity adds an additional parameter $ntasks$, the number of subtasks (number of distinct versions of sparse parity) present in the dataset. To construct the task, we first choose $ntasks$ random subsets S_i of k indices from $\{1, 2, \dots, n\}$: $S_i \subset \{1, 2, \dots, n\}$ and $|S_i| = k$, where $i = 1, 2, \dots, ntasks$. Input bit strings are length $ntasks + n$. We call the first $ntasks$ bits the *control bits* and the last n bits the *task bits*. If control bit i is active, then the parity is computed from the subset S_i of the task bits. The control bits 1-hot encode the task number: on a given input, only one control bit is set to 1 at a time – the rest are zero. For the sample shown below, since control bit 2 is active, the answer is the parity of the task bits $S_2 = \{2, 7\}$, which is 0 for this input:



We impose a uniform distribution over the task bits. On the control bits, we impose a Zipfian distribution: the probability that a sample has control bit i active (and therefore the parity must be computed from the subset S_i of the task bits) is $\frac{1}{Z}i^{-(\alpha+1)}$ where $Z = \sum_{i=1}^{ntasks} i^{-(\alpha+1)}$. This imposes a power law distribution over subtasks in data. Since answers are parities, this task can be treated as a binary classification problem on the subset of bit strings $\{0, 1\}^{ntasks+n}$ where for each string all but one bit of the first $ntasks$ bits are zero.

2.3.2 Power law scaling and emergence

We train ReLU MLPs with a single hidden layer to solve this task with cross-entropy loss. The input dimension is $ntasks + n$. We use the Adam optimizer with a learning rate of 10^{-3} . To study scaling with respect to the number of model parameters, we train networks of varying width by sampling batches online. Within an individual single-epoch training run, we can study scaling in steps S . To study scaling with respect to multi-epoch training dataset size D , we use a network of sufficient width for capacity to not be a bottleneck, and for varying D we sample a training set of D samples and train for multiple epochs, recording model performance when mean test loss is lowest (early-stopping).

Training dynamics on the multitask sparse parity problem are highly nontrivial – on each individual subtask, loss follows a reverse-S curve, dropping after an initial plateau. This transition

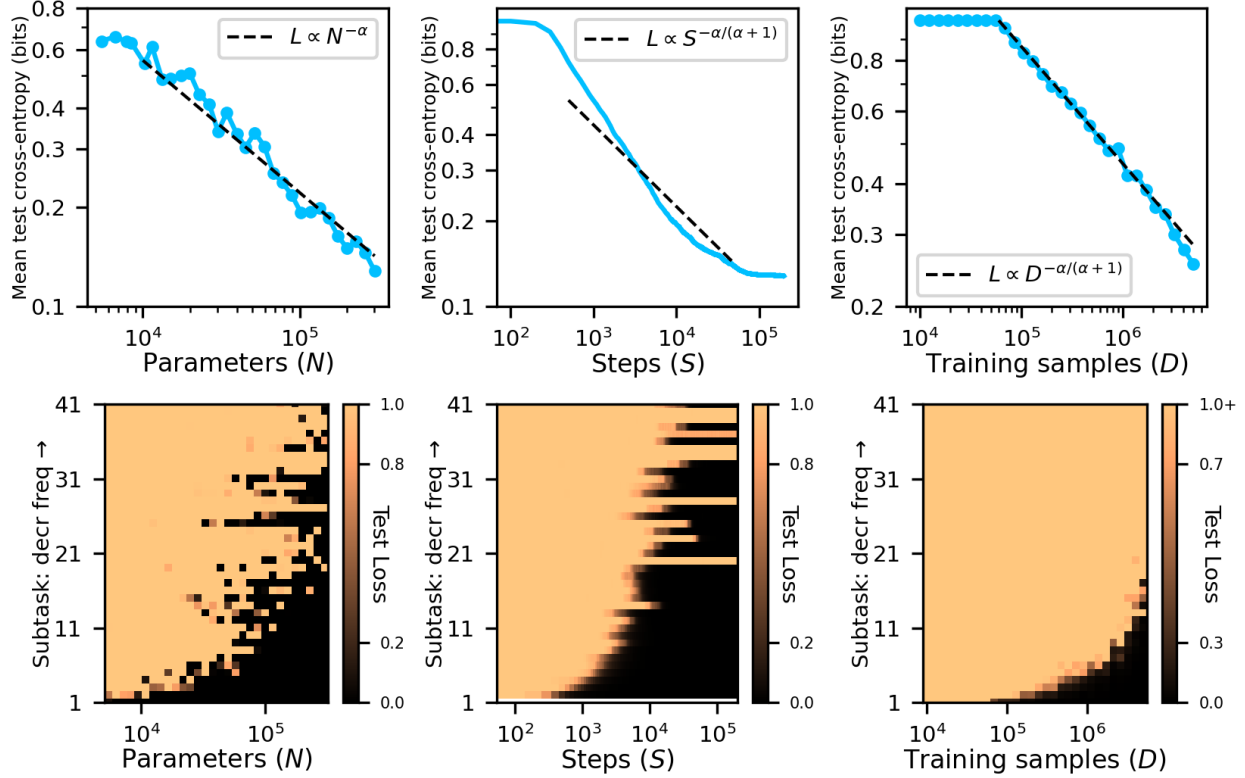


Figure 2.2: **Top:** Neural networks exhibit power law scaling in loss w.r.t. parameters N , training time S , and training samples D (for multi-epoch training) when trained on the multitask sparse parity dataset. Here $\alpha = 0.4$ and we plot lines $\propto N^{-\alpha}$, $\propto S^{-\alpha/(\alpha+1)}$, $\propto D^{-\alpha/(\alpha+1)}$. **Bottom:** neural scaling broken down by subtask. Scaling behavior on individual subtasks exhibits emergence, where subtasks are suddenly learned above a particular scale. Power law neural scaling of mean test loss averages over a large number of qualitative changes in network performance (when broken down by subtask), with loss being driven to zero on an increasing number of subtasks which are power law distributed in frequency, a realization of the mechanism of neural scaling discussed in Section 2.2.

happens at different times for different subtasks, so the overall loss decreases smoothly, averaging over these transitions. See Section 2.8.2 for more discussion of training dynamics.

Figure 2.2 shows scaling curves on the multitask sparse parity problem. For the results shown, we used $n_{\text{tasks}} = 500$, $n = 100$, $k = 3$, $\alpha = 0.4$, and a batch size of 20000. We vary training dataset size from $1e4$ to $5e6$ and vary hidden-layer width from 10 to 500 neurons. We train for $2e5$ steps. In line with the theory from Section 2.2, we find that as we scale training data and parameters, networks learn more and more quanta (reducing loss on more and more subtasks), roughly in order of their frequency, and that this is what drives neural scaling. We see that scaling w.r.t. parameters is noisier than data scaling, possibly due to model initialization having some influence on which quanta are learned (for our data scaling experiments, we use the same seed and same model size for all runs). We also see that for scaling on individual subtasks, there is a rough scale of data or parameters below which networks do not learn the task, and above which they do. Smooth power law scaling therefore averages over a large number of emergent changes in model

performance when properly decomposed by subtask, a proof of concept that the Quantization Model can be the mechanism of neural scaling for data with the right structure. See Section 2.8.2 for additional results and discussion on how the scaling exponents α_N , α_S , α_D relate to the subtask distribution power law exponent $\alpha + 1$ empirically.

2.4 Decomposing LLM scaling laws

We now study how scaling curves for large language models decompose. For our experiments, we use the Pythia model suite from Eleuther AI [29], a set of decoder-only transformers of varying size trained on approximately 300 billion tokens of The Pile [30]. We evaluate several models in the suite (ranging from 19m to 6.4b non-embedding parameters) on approximately 10 million tokens from the test set of The Pile. We record cross-entropy loss on every token, enabling us to study how loss on individual tokens, as well as how the distribution over losses, changes with model scale.

2.4.1 The distribution over per-token losses

In Figure 2.3, we show how the distribution over losses scales with model size. First, we find that for the first six models in the Pythia sequence, the mean loss scales as a power law with exponent $\alpha_N = 0.083$, roughly in line with the parameter scaling exponent of 0.076 measured in [3]. The 6.4b model does not fit the scaling curve well, so we excluded its loss when measuring the scaling exponent. Next, we plot the probability distribution over per-token losses $p(L)$. We find that losses close to zero are by far the most common, and that scaling increases the portion of approximately-zero losses. We also plot $Lp(L)$, the probability density over losses weighted by loss. The mean loss is the area under this curve. We see that despite approximately-zero-loss tokens being by far the most common, they do not contribute much mass to the mean loss. See Figure 2.11 for how these distributions change over training steps rather than across model size. We note that neural scaling in the wild is much more complicated than for multitask sparse parity – notably, the distribution over losses is not bimodal. We leave a detailed study of whether the statistics of neural scaling in LLMs are compatible with prior models of neural scaling to future work.

2.4.2 Monogenic versus polygenic scaling curves

In our introduction of the Quantization Hypothesis in Section 2.2 and our multitask sparse parity study in Section 2.3 we modeled network performance on individual samples as benefitting from a single quantum – all samples belong to a single subtask, which is either solved or not solved in a binary fashion. In our model and on multitask sparse parity, scaling curves on individual examples all exhibit emergence – loss on individual examples undergoes a sharp transition at a particular scale of parameters or data. Do we observe this in large language models?

Inspecting a large number of per-token (per-sample) scaling curves, we observe a variety of scaling behaviors. On some samples, loss drops at a particular scale. More typically though, loss improves at multiple scales. If the Quantization Hypothesis is true and the effect of scaling is to simply add new quanta to the model, then for per-sample loss curves to show progress at multiple scales, those samples must benefit from multiple quanta additively. As first mentioned

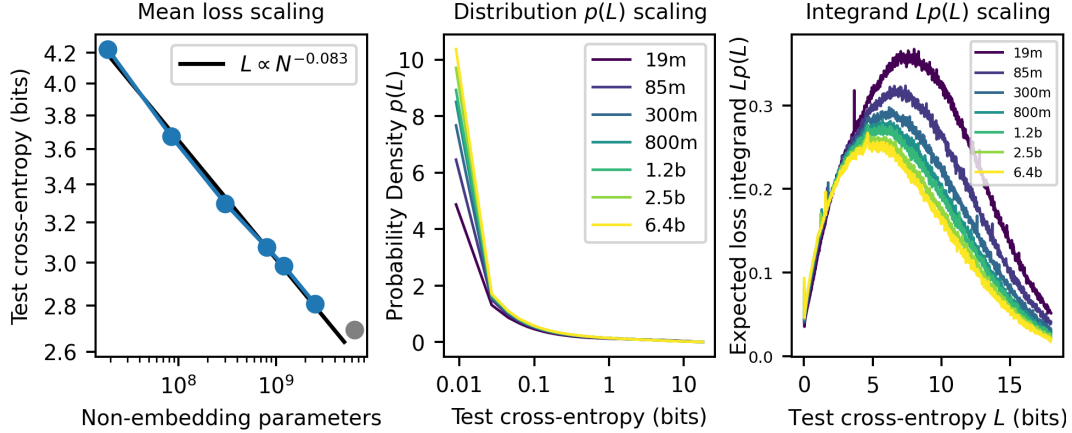


Figure 2.3: **Left:** Scaling of mean test loss w.r.t. non-embedding parameters for the Pythia models [29]. The parameter scaling exponent α_N is measured to be ≈ 0.083 from the first six points along the curve (the seventh model appears to break the trend). **Center:** the distribution $p(L)$ over losses on individual samples for models of different size. Losses ≈ 0 are by far the most common, and larger models achieve ≈ 0 loss on an increasing fraction of samples. **Right:** the expected loss integrand $Lp(L)$ for models of different sizes. Low-loss samples contribute minimal mass to the mean loss, which is instead dominated by samples with much higher loss of 5-10 bits (depending on scale).

in Section 2.2, we borrow terminology from genetics and refer to prediction problems for which the model’s performance is determined by a single quantum as *monogenic* (akin to when a single gene determines a trait) and as *polygenic* when multiple quanta influence performance (in analogy to when multiple genes contribute to a trait). In multitask sparse parity, all prediction problems are monogenic. In natural language, we observe that model performance on most tokens improves at multiple scales, suggesting that most tokens are polygenic, but we can find tokens for which loss drops as a single phase transition in scale. Polygenicity forms a spectrum: the smoothness of the loss curve can vary substantially between examples, presumably with some prediction problems using few quanta and others using many. In Figure 2.4, we show extreme examples of both monogenic and polygenic samples.

Note that our monogenic/polygenic taxonomy of model behaviors assumes that QH1 and QH2 are true. However, it could be the case that there isn’t an underlying discreteness to what is learned, or that scaling totally changes what networks learn, rather than simply adding additional quanta. Whether scaling truly has the effect we described will have to be investigated in future studies of the internals of neural networks. We also note that it is possible that sharp transitions in the per-token loss curves could be due to noise – if we had multiple runs with different random seeds for each model scale, we could better test whether the mean loss across seeds decreases smoothly or if there is a genuine discreteness where gradual progress is impossible for apparently “monogenic” tokens.

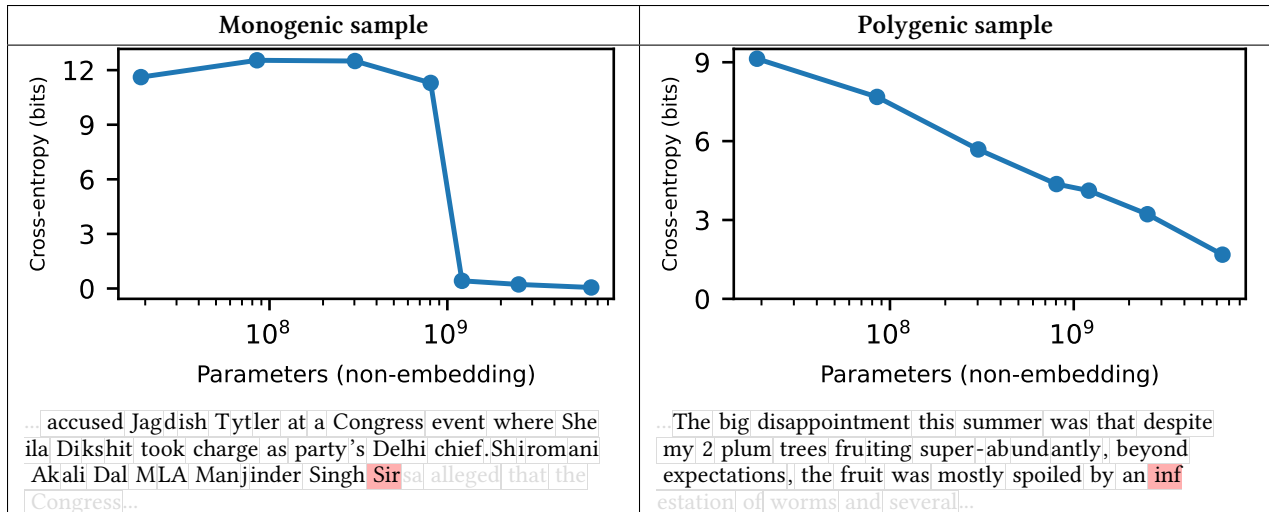


Figure 2.4: Per-sample scaling curves can have diverse behavior. Here we show extreme examples where scaling (of loss on predicting the highlighted token) is abrupt versus smooth. If the Quantization Hypothesis describes language modeling, then samples with sharp scaling would be *monogenic*, displaying sharp emergence at a particular model scale when the relevant quantum is learned. Samples with gradual scaling would be *polygenic*, where many quanta, emerging at different scales, marginally improve the loss. We show additional examples in Figure 2.12.

2.5 The quanta of language modeling

We have conjectured that the internals and behavior of language models are decomposable into an enumerable set of modules and associated skills (quanta). What might these basic building blocks of LLMs be? In this section, we develop a preliminary method to discover quanta. In particular, we will attempt to cluster tokens in a language corpus according to what knowledge or skill LLMs use to predict those tokens from their context. Our goal is to find coherent clusters of language model behavior that each reveal some distinct skill that the model has learned. Note that in *clustering* tokens to discover quanta, we are making the likely unrealistic assumption that these tokens are monogenic – that there is only one quantum involved in predicting each token. Not also that these clusters of behavior will not give us a mechanistic understanding of the quanta, but simply provide examples of LLM skills which could be studied further in future work.

We propose the use of gradients to cluster next-token prediction samples, where a “sample” consists of a token and its context in some document. Given some model, we will cluster two samples together if the gradient of the model’s loss on each sample w.r.t. the model’s parameters is similar for the two samples. The intuition for using gradients is as follows: if a model uses the same internal module to generate its prediction on two samples, then the gradients for parameters within the module may be nonzero and similar for the two samples (and possibly ≈ 0 for parameters in irrelevant modules). If a model uses different modules to generate its prediction on different samples, then the gradients may not overlap. We therefore use gradient similarity as a proxy for *mechanistic similarity* – whether a model uses similar mechanisms/modules to generate its prediction on distinct samples. While crude, we find that gradients contain enough information to allow us to automatically discover many coherent clusters of LLM behavior using the following

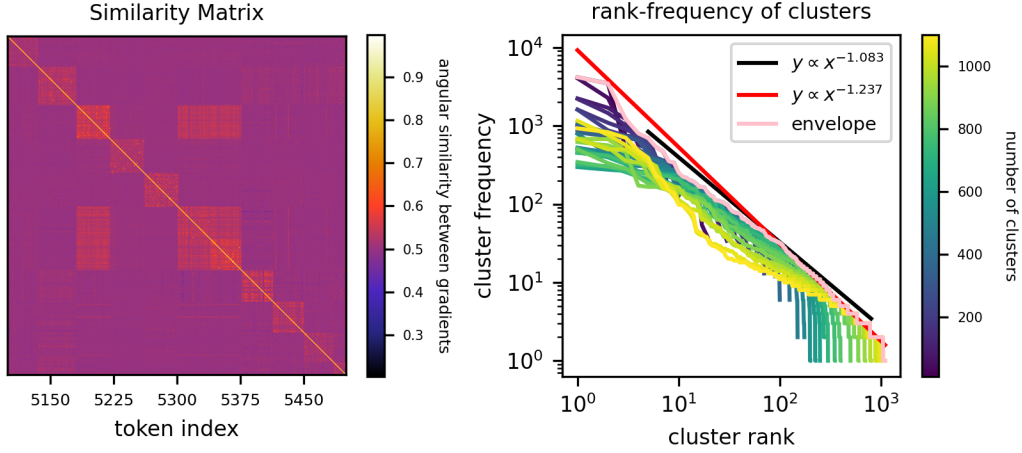


Figure 2.5: **Left:** angular similarity between model gradients for a variety of natural language samples. Samples are reordered according to their QDG cluster (with 400 clusters) to reveal the block-diagonal structure of the similarity matrix. We visualize a small part of the overall similarity matrix in this plot – note that not all clusters are as visibly distinct as the ones shown. **Right:** rank-frequency plot of QDG clusters. We measure the slope of the envelope of the rank-frequency curves from cluster rank 100-1000 to be ≈ -1.24 , which is a steeper than the slope of -1.08 expected from the measured parameter-scaling exponent from Figure 2.3, though within the margin of error given the uncertainty of our clustering methodology. See Section 2.8.5 for a discussion of the bias/uncertainty of our method.

algorithm:

Quanta Discovery from Gradients (QDG): We will use spectral clustering on gradients to find clusters of samples whose gradient has nonzero cosine similarity. Given a set of samples (x_i, y_i) and a model f_θ , we compute gradients for each sample $g_i = \nabla_\theta L(f_\theta(x_i), y_i)$. We then normalize these gradients $g_i \mapsto \hat{g}_i$ so that $\hat{g}_i \cdot \hat{g}_i = 1$. Let A be a matrix whose rows are the normalized gradients: $A_{i,\cdot} = \hat{g}_i$. If we are clustering d samples and our model has n parameters, A has shape (d, n) . We compute an affinity matrix $C = AA^T$, a matrix of shape (d, d) where $C_{ij} = \hat{g}_i \cdot \hat{g}_j$, the cosine similarity between gradients g_i, g_j . From this, we compute an affinity matrix of the angular similarities $\hat{C}$ (which take values in $[0, 1]$) via $\hat{C}_{ij} = 1 - \arccos(C_{ij})/\pi$. We then perform spectral clustering with $\hat{C}$ to cluster samples.

QDG is expensive to compute for large models and for large numbers of samples. We therefore only apply it to the smallest model in the Pythia suite, which has 19m non-embedding parameters. We cluster 10000 tokens on which this model is confident and correct in its prediction, achieving less than 0.1 nats of cross-entropy. See Section 2.8.3 for more detail.

We find that many, though not all, QDG clusters reveal some coherent model behavior. We show examples from clusters in Figure 2.1 and Figure 2.13. These clusters were found with the spectral clustering hyperparameter `n_clusters = 400`. While most clusters involve the prediction of the same token, manually inspecting these clusters we find that they usually involve predicting the same token for a coherent reason, rather than being based merely on having the same output. We also find clusters for more abstract prediction rules. For instance, the quantum shown on the left column of Figure 2.1 is the skill of incrementing a numerical sequence, and the

examples involve predicting a variety of different tokens representing numbers.

2.5.1 The natural distribution over language modeling quanta

In our model, some quanta are more frequently used than others. If these frequencies follow a power law in accordance with the Quantization Hypothesis, then we may expect QDG cluster sizes to be governed by a power law. The measured scaling exponent of $\alpha_N = 0.083$ from Figure 2.3 implies a power law distribution over quanta with exponent -1.083 . Do the cluster sizes follow this?

Figure 2.5 shows rank-frequency curves for clusters discovered with QDG for varying choices of `n_clusters`. These curves sort the clusters according to their size and then plot size against cluster index (rank). We plot rank-frequency curves for many choices of `n_clusters` since it is unclear a priori which `n_clusters` to use. When we measure the slope of the rank-frequency curve, we measure it from the envelope formed by the many rank-frequency curves, a practice which we discuss in Section 2.8.5. Biases in the clustering algorithm and inherent noise in model gradients make clustering imperfect, and lead to high uncertainty of our the measured power law exponent. From our analysis in Section 2.8.5, we think that extracting the power law exponent over quanta utilization frequency by measuring the slope of the rank-frequency curve should have uncertainty of at least 0.2. We also note that some rank-frequency curves don’t look like a clean power law. In Section 2.8.5, Figure 2.16 we find that we can get similar-looking curves in a toy model of this clustering process when the dimension and noise is high. Between ranks 100-1000, we measure a slope of ≈ -1.24 , about 0.16 off our expected slope of -1.08 , and so within the margin of error. We are encouraged that the size of our discovered clusters seem to decay at a rate (very roughly) compatible with observed neural scaling exponents, in line with our theory. However, less naive clustering schemes, operating on more samples with more clusters, could be useful to sharpen this measurement.

2.6 Related Work

Models of neural scaling: Several models of neural scaling laws have been proposed in prior work. Sharma and Kaplan [31] explain power law scaling w.r.t. model parameters using an argument from approximation theory, which relates neural scaling exponents to the dimension of the data manifold d . Michaud, Liu, and Tegmark [32] point out that effective dimension d could be generalized to the maximum arity of the target function’s computation graph for sparse compositional problems. Bahri et al. [33] generalized the model of Sharma and Kaplan [31] to scaling w.r.t. dataset size, additionally relating scaling exponents to the power law spectrum of certain kernels. Maloney, Roberts, and Sully [34] develop an exactly solvable random-feature model of scaling, from which they derive a joint parameter-data scaling law. Bordelon, Canatar, and Pehlevan [35] develop a model of data scaling for kernels, decomposing the generalization error into a sum over eigenmodes, whereas we decompose error into a sum over quanta. Arguably the closest prior work to ours is Hutter [36], who develops a model of data scaling wherein a discrete set of “features” must be learned. In this model, a feature is learned if it occurs at least once in the training set. If the features are Zipfian distributed, this produces power law scaling in expectation but with high variance. In our model, using a data threshold $\tau \gg 1$ lowers the

variance in the scaling curve, and we also considered scaling w.r.t. parameters and applied the model to real networks.

Understanding emergent abilities: Wei et al. [8] and Srivastava et al. [37] document examples of emergent abilities in large language models, though Schaeffer, Miranda, and Koyejo [38] suggest that these examples are an artifact of the metric used to evaluate performance. Arora and Goyal [39] develop a framework for the emergence of “skills”, where predicting text requires combining multiple different skills from an underlying set of language skills.

Miscellaneous: The topic of phase transitions in machine learning is not new [40], but our work was strongly influenced by the recent work of Olsson et al. [25] who observe a phase change from the formation of induction heads and especially Nanda et al. [13] who conjecture that phase changes may be ubiquitous. Simon et al. [41] also exhibit a task where learning proceeds as a series of discrete steps. Chen et al. [42] develop a framework for understanding LLM “skills” in a hierarchy and for choosing data to more efficiently learn desired skills. Chan et al. [43] study how a Zipfian data distribution influences in-context learning.

2.7 Discussion

Summary: The Quantization Hypothesis posits that for some types of prediction problems, models must learn a discrete (quantized) set of modules/knowledge/skills (quanta). When data is distributed in such a way that the “use frequencies” of these quanta follow a power law, then power law neural scaling can arise as models learn more and more quanta, with smooth scaling curves averaging over many small cases of emergence. We presented a toy dataset where neural scaling exhibits these properties. We then documented how language model scaling curves decompose, beyond simply how the mean loss scales. Lastly, we developed a method to discover quanta from the internal structure of trained models, from which we were able to enumerate a large number of skills of a small language model. The frequencies at which the quanta we discover are used for prediction in natural text seem to roughly track the power law our theory would predict, though this measurement is quite imprecise.

Limitations: While the Quantization Hypothesis appears to hold for our toy datasets, much work remains in investigating to what extent it holds for natural tasks like language modeling. Probably our riskiest assumption was that there is an underlying discreteness to *everything* that models learn. Gradual scaling seems typical in LLMs [38], and it could be more parsimonious to model neural scaling as an underlying smooth process rather than to assume that most tasks are highly polygenic with underlying discrete quanta. Note also that in our model of scaling w.r.t. parameters N , having more parameters merely increases the capacity of the network. In practice however, larger networks are more efficient learners [7], and one can trade off between parameters and data, whereas in our model parameters and data independently bottleneck the number of quanta that can be learned. Additionally, we modeled the quanta as being independent, where learning order is given just by the use frequencies, but it could make more sense to think of the quanta as living in a hierarchical dependency graph. Lastly, our QDG method is neither very principled nor scalable, and much better methods could likely be developed to discover quanta and study their statistics for larger models and across more samples.

Implications for emergence and forecasting: Srivastava et al. [37] find that on some tasks, neural scaling has high “linearity”, with gradual improvements to scale, with other tasks displaying

“breakthroughness”, where performance improves sharply at some scale. In our model, high linearity would result from a task’s relevant quanta being widely spread along the Q Sequence, and high breakthroughness would result from a task being monogenic or from the relevant quanta being close together in the Q Sequence. Our model also suggests that future capabilities could be forecasted if one could estimate the frequency at which that skill would benefit prediction in the training corpus.

Implications for mechanistic interpretability: If the Quantization Hypothesis is correct, then understanding a network reduces to enumerating its quanta. Having done this, the quanta could perhaps then be translated into a more interpretable format (something like code), studied in this format, and eventually executed in this format, rather than via the operation of the network.

Outlook: Lastly, our decomposition of networks into quanta is reminiscent of Minsky’s *Society of Mind* [44] perspective that minds are decomposable into individually mindless “agents”. If this decomposition is indeed possible, then the quanta (agents) become natural objects of study within networks. This *mesoscale* understanding of networks, in terms of the internal modules which collectively constitute their performance, could perhaps act like statistical physics for deep learning, allowing us to bridge our microscale understanding of low-level training dynamics and our macroscale understanding of model performance.

Acknowledgements

We thank Tamay Besiroglu, Neel Nanda, Tony Wang, David Bau, Ben Edelman, Brian Cheung, Wes Gurnee, Stephen Casper, Peter Hase, Davis Brown, Eleni Shor, Max Nadeau, and Xander Davies for helpful conversations and feedback. We thank Lauro Langosco for helping with code to visualize samples from The Pile. This work was supported by the Foundational Questions Institute, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), and IAIFI through NSF grant PHY-2019786.

References

- [1] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. Patwary, M. Ali, Y. Yang, and Y. Zhou. “Deep learning scaling is predictable, empirically.” *arXiv preprint arXiv:1712.00409*, 2017.
- [2] J. S. Rosenfeld, A. Rosenfeld, Y. Belinkov, and N. Shavit. “A constructive prediction of the generalization error across scales.” *arXiv preprint arXiv:1909.12673*, 2019.
- [3] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling laws for neural language models.” *arXiv preprint arXiv:2001.08361*, 2020.
- [4] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T. B. Brown, P. Dhariwal, S. Gray, et al. “Scaling laws for autoregressive generative modeling.” *arXiv preprint arXiv:2010.14701*, 2020.

- [5] M. A. Gordon, K. Duh, and J. Kaplan. “Data and parameter scaling laws for neural machine translation.” In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021, pp. 5915–5922.
- [6] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer. “Scaling vision transformers.” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12104–12113.
- [7] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. “Training compute-optimal large language models.” *arXiv preprint arXiv:2203.15556*, 2022.
- [8] J. Wei et al. “Emergent Abilities of Large Language Models.” *Transactions on Machine Learning Research*, 2022. Survey Certification. ISSN: 2835-8856. URL: <https://openreview.net/forum?id=yzkSU5zdwD>.
- [9] J. Steinhardt. *Future ML Systems Will Be Qualitatively Different*. Accessed: 2023-10-26. Jan. 2022. URL: <https://bounded-regret.ghost.io/future-ml-systems-will-be-qualitatively-different/>.
- [10] D. Ganguli, D. Hernandez, L. Lovitt, A. Askell, Y. Bai, A. Chen, T. Conerly, N. Dassarma, D. Drain, N. Elhage, et al. “Predictability and surprise in large generative models.” In: *2022 ACM Conference on Fairness, Accountability, and Transparency*. 2022, pp. 1747–1764.
- [11] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. “Zoom In: An Introduction to Circuits.” *Distill*, 2020. <https://distill.pub/2020/circuits/zoom-in>. DOI: [10.23915/distill.00024.001](https://doi.org/10.23915/distill.00024.001).
- [12] N. Cammarata, G. Goh, S. Carter, L. Schubert, M. Petrov, and C. Olah. “Curve detectors.” *Distill*, 5(6), 2020, e00024–003.
- [13] N. Nanda, L. Chan, T. Liberman, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.
- [14] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. “Emergent world representations: Exploring a sequence model trained on a synthetic task.” *arXiv preprint arXiv:2210.13382*, 2022.
- [15] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt. “Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small.” *arXiv preprint arXiv:2211.00593*, 2022.
- [16] N. Elhage et al. “A Mathematical Framework for Transformer Circuits.” *Transformer Circuits Thread*, 2021. <https://transformer-circuits.pub/2021/framework/index.html>.
- [17] T. Lieberum, M. Rahtz, J. Kramár, G. Irving, R. Shah, and V. Mikulik. “Does circuit analysis interpretability scale? evidence from multiple choice capabilities in chinchilla.” *arXiv preprint arXiv:2307.09458*, 2023.
- [18] S. Casper, S. Hod, D. Filan, C. Wild, A. Critch, and S. Russell. “Graphical clusterability and local specialization in deep neural networks.” In: *ICLR 2022 Workshop on PAIR { \textasciicircum } 2Struct: Privacy, Accountability, Interpretability, Robustness, Reasoning on Structured Data*. 2022.

- [19] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. URL: <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [20] J. Frankle and M. Carbin. “The lottery ticket hypothesis: Finding sparse, trainable neural networks.” *arXiv preprint arXiv:1803.03635*, 2018.
- [21] D. Bayazit, N. Foroutan, Z. Chen, G. Weiss, and A. Bosselut. “Discovering Knowledge-Critical Subnetworks in Pretrained Language Models.” *arXiv preprint arXiv:2310.03084*, 2023.
- [22] Y. Li, J. Yosinski, J. Clune, H. Lipson, and J. Hopcroft. *Convergent Learning: Do different neural networks learn the same representations?* arXiv:1511.07543 [cs]. Feb. 2016. URL: <http://arxiv.org/abs/1511.07543> (visited on 08/19/2022).
- [23] T. Nguyen, M. Raghu, and S. Kornblith. *Do Wide and Deep Networks Learn the Same Things? Uncovering How Neural Network Representations Vary with Width and Depth*. arXiv:2010.15327 [cs]. Apr. 2021. URL: <http://arxiv.org/abs/2010.15327>.
- [24] A. Dravid, Y. Gandelsman, A. A. Efros, and A. Shocher. *Rosetta Neurons: Mining the Common Units in a Model Zoo*. arXiv:2306.09346 [cs]. June 2023. DOI: [10.48550/arXiv.2306.09346](https://doi.org/10.48550/arXiv.2306.09346). URL: <http://arxiv.org/abs/2306.09346>.
- [25] C. Olsson et al. “In-context Learning and Induction Heads.” *Transformer Circuits Thread*, 2022. <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [26] G. Branwen. *The scaling hypothesis*. 2021. URL: <https://gwern.net/scaling-hypothesis>.
- [27] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra. “Grokking: Generalization beyond overfitting on small algorithmic datasets.” *arXiv preprint arXiv:2201.02177*, 2022.
- [28] B. Barak, B. L. Edelman, S. Goel, S. Kakade, E. Malach, and C. Zhang. “Hidden progress in deep learning: Sgd learns parities near the computational limit.” *arXiv preprint arXiv:2207.08799*, 2022.
- [29] S. Biderman, H. Schoelkopf, Q. Anthony, H. Bradley, K. O’Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, et al. “Pythia: A suite for analyzing large language models across training and scaling.” *arXiv preprint arXiv:2304.01373*, 2023.
- [30] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. “The pile: An 800gb dataset of diverse text for language modeling.” *arXiv preprint arXiv:2101.00027*, 2020.
- [31] U. Sharma and J. Kaplan. “Scaling Laws from the Data Manifold Dimension.” *Journal of Machine Learning Research*, 23(9), 2022, pp. 1–34. URL: <http://jmlr.org/papers/v23/20-1111.html>.
- [32] E. J. Michaud, Z. Liu, and M. Tegmark. “Precision Machine Learning.” *Entropy*, 25(1), 2023. ISSN: 1099-4300. DOI: [10.3390/e25010175](https://doi.org/10.3390/e25010175). URL: <https://www.mdpi.com/1099-4300/25/1/175>.
- [33] Y. Bahri, E. Dyer, J. Kaplan, J. Lee, and U. Sharma. “Explaining neural scaling laws.” *arXiv preprint arXiv:2102.06701*, 2021.
- [34] A. Maloney, D. A. Roberts, and J. Sully. “A Solvable Model of Neural Scaling Laws.” *arXiv preprint arXiv:2210.16859*, 2022.

- [35] B. Bordelon, A. Canatar, and C. Pehlevan. “Spectrum dependent learning curves in kernel regression and wide neural networks.” In: *International Conference on Machine Learning*. PMLR, 2020, pp. 1024–1034.
- [36] M. Hutter. “Learning curve theory.” *arXiv preprint arXiv:2102.04074*, 2021.
- [37] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoen, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, et al. “Beyond the imitation game: Quantifying and extrapolating the capabilities of language models.” *arXiv preprint arXiv:2206.04615*, 2022.
- [38] R. Schaeffer, B. Miranda, and S. Koyejo. “Are Emergent Abilities of Large Language Models a Mirage?” *arXiv preprint arXiv:2304.15004*, 2023.
- [39] S. Arora and A. Goyal. “A theory for emergence of complex skills in language models.” *arXiv preprint arXiv:2307.15936*, 2023.
- [40] L. Saitta, A. Giordana, and A. Cornuejols. *Phase transitions in machine learning*. Cambridge University Press, 2011.
- [41] J. B. Simon, M. Knutins, L. Ziyin, D. Geisz, A. J. Fetterman, and J. Albrecht. “On the stepwise nature of self-supervised learning.” *arXiv preprint arXiv:2303.15438*, 2023.
- [42] M. F. Chen, N. Roberts, K. Bhatia, J. Wang, C. Zhang, F. Sala, and C. Ré. “Skill-it! A Data-Driven Skills Framework for Understanding and Training Language Models.” *arXiv preprint arXiv:2307.14430*, 2023.
- [43] S. Chan, A. Santoro, A. Lampinen, J. Wang, A. Singh, P. Richemond, J. McClelland, and F. Hill. “Data distributional properties drive emergent in-context learning in transformers.” *Advances in Neural Information Processing Systems*, 35, 2022, pp. 18878–18891.
- [44] M. Minsky. *Society of mind*. Simon and Schuster, 1988.
- [45] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python.” *Journal of Machine Learning Research*, 12, 2011, pp. 2825–2830.
- [46] R. Eldan and Y. Li. “TinyStories: How Small Can Language Models Be and Still Speak Coherent English?” *arXiv preprint arXiv:2305.07759*, 2023.
- [47] P. Villalobos. *Scaling Laws Literature Review*. <https://epochai.org/blog/scaling-laws-literature-review>. Accessed: 2023-01-31. 2023.
- [48] J. Droppo and O. Elilol. “Scaling laws for acoustic models.” *arXiv preprint arXiv:2106.09488*, 2021.
- [49] N. Ardalani, C.-J. Wu, Z. Chen, B. Bhushanam, and A. Aziz. “Understanding Scaling Laws for Recommendation Models.” *arXiv preprint arXiv:2208.08489*, 2022.

2.8 Appendix

2.8.1 More general scaling laws

If one learns the first n quanta, reducing the loss from b_k to a_k ($1 \leq k \leq n$), while the loss remains b_k for $k > n$, the expected loss is given by:

$$L_n = \sum_{k=1}^n a_k p_k + \sum_{k=n+1}^{\infty} b_k p_k. \quad (2.3)$$

In the main text, we used $a_k = a$ and $b_k = b$ for our model. However, one can imagine a variety of other choices for a_k and b_k .

Case 1 $b_k = -\log p_k$ and $a_k = 0$, where $p_k = k^{-(\alpha+1)}/\zeta(\alpha+1)$. This baseline for b_k is the error of a model which outputs the token frequencies, independent of the context (assuming that quanta involve the prediction of a particular token). The expected loss is given by:

$$L_n = \sum_{k=1}^n 0 \cdot p_k + \sum_{k=n+1}^{\infty} (-\log p_k) \cdot p_k \approx \frac{1 + \alpha + \alpha \log \zeta(\alpha+1)}{\alpha^2 \zeta(\alpha+1)} n^{-\alpha} + \frac{\alpha+1}{\alpha \zeta(\alpha+1)} n^{-\alpha} \log n, \quad (2.4)$$

which contains a power law term $n^{-\alpha}$ plus a log term $n^{-\alpha} \log n$. For very large n , the log term can be ignored, so L is still approximately a power law of n with exponent $-\alpha$, shown in Figure 2.6.

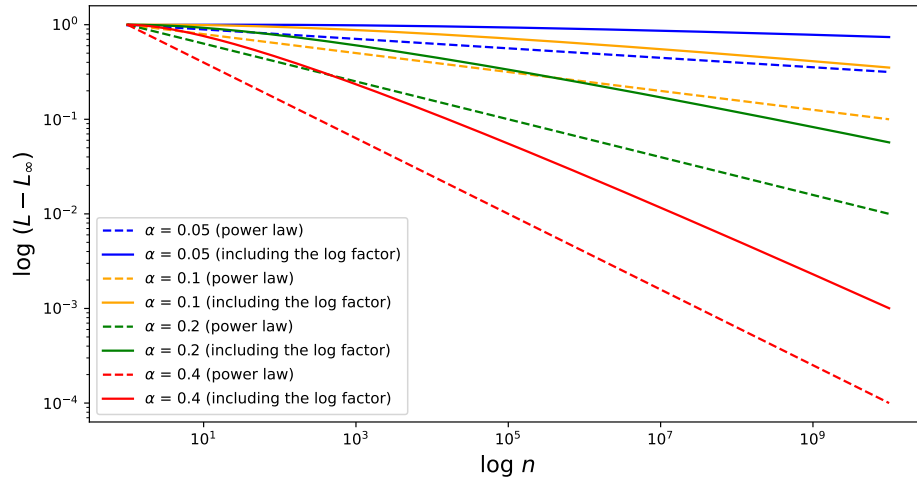


Figure 2.6: Comparing different scaling laws. Setting $a_k = 0$, we compare $b_k = -\log p_k$ (solid lines) and $b_k = 1$ (dashed lines) for different alphas. Although the $b_k = -\log p_k$ case would cause an extra loss term $n^{-\alpha} \log n$ in addition to the power law term $n^{-\alpha}$, the loss becomes a power law asymptotically when n becomes large.

Case 2 $b_k = -\log p_k$ and $a_k = -\log(Cp_k)$ ($C > 1$), where $p_k = k^{-(\alpha+1)}/\zeta(\alpha+1)$. The expected loss is given by:

$$L_n = \sum_{k=1}^n (-\log(Cp_k)) \cdot p_k + \sum_{k=n+1}^{\infty} (-\log p_k) \cdot p_k \approx \frac{\log C}{\alpha \zeta(\alpha+1)} n^{-\alpha} - \log C + \frac{1 + \alpha + \alpha \log \zeta(\alpha+1)}{\alpha^2 \zeta(\alpha+1)}, \quad (2.5)$$

which is a power law $n^{-\alpha}$ plus a constant.

2.8.2 Additional results on multitask sparse parity

Training dynamics: When loss is broken down by subtask on multitask sparse parity, learning curves consist of many reverse-S shaped curves, and mean loss decreases smoothly as an average over these curves. In Figure 2.7, we show loss versus time for each subtask for training runs in both the single-epoch and multi-epoch regimes. In Figure 2.8 we show how convergence time for each subtask relates to the frequency of that subtask.

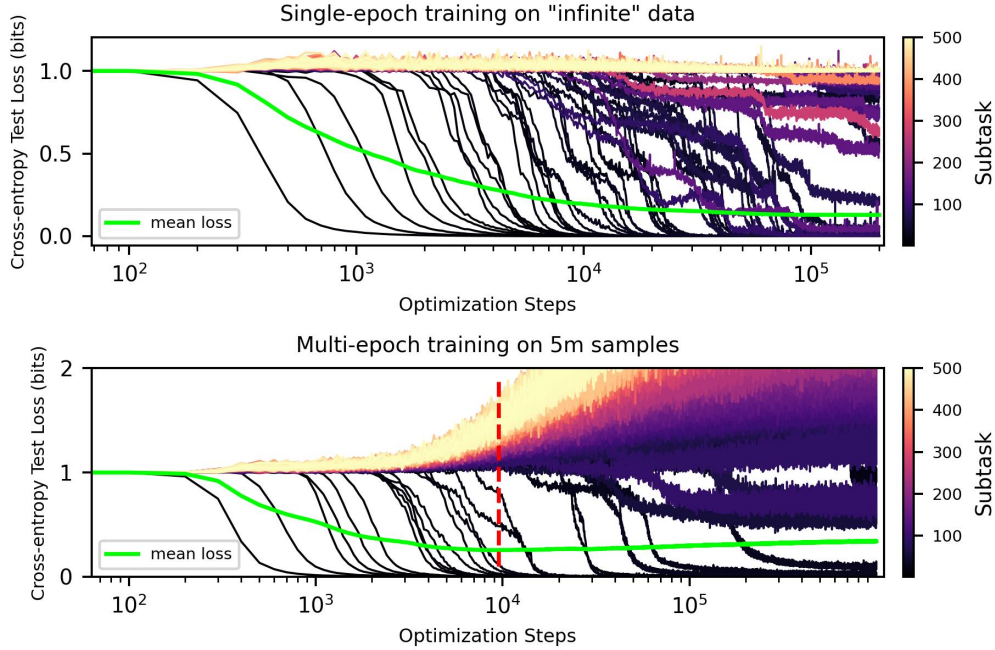


Figure 2.7: Training dynamics on the multitask sparse parity dataset consist of many “phase transitions” when decomposed by subtask – the loss curve for each subtask drops following an initial plateau of no apparent progress, in line with [28]. The mean loss decreases smoothly, averaging over these phase transitions in the model’s performance on subtasks. We show curves for single-epoch training (top) and multi-epoch training on 5 million samples (bottom). The dashed red line indicates the early stopping point where mean test loss is minimized. For these runs, $\alpha = 0.4$.

Scaling for varying α : In Figure 2.10 we show scaling curves on multitask sparse parity in N, S, D for a variety of quanta distribution parameters α . While all scaling curves appear to be power laws, the relationship between $\alpha_N, \alpha_S, \alpha_D$ and α is not precisely as predicted by theory:

1. **Parameter scaling:** We observe that the relationship between α_N and α deviates a bit from the prediction $\alpha_N = \alpha$, with $\alpha_N < \alpha$ for small α and $\alpha_N > \alpha$ for large α . Perhaps model size does not influence learning just by changing capacity, but also by affecting optimization.
2. **Step scaling:** We observe that α_S is consistently higher than the theoretical prediction $\alpha/(\alpha + 1)$. In Figure 2.8, we saw that the number of steps to convergence for each subtask did not precisely follow $S_k \propto p_k^{-1}$, but was closer to $S_k \propto p_k^{-0.81}$. This means that many subtasks converge faster than we would expect, producing a steeper scaling curve.

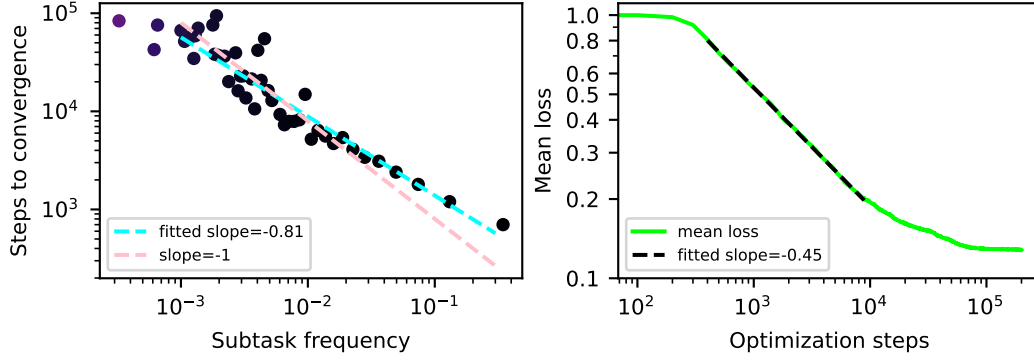


Figure 2.8: Convergence time for each subtask versus the frequency of that subtask. We see that convergence time S_k on subtask k is $S_k \propto p_k^{-0.81}$ rather than $S_k \propto p_k^{-1}$ as we had expected. This leads to a steeper scaling w.r.t. S than expected from theory. For these experiments, we used $\alpha = 0.4$, and so we would have predicted $\alpha_S \approx 0.29$ but instead we get $\alpha_S \approx 0.45$. We consider the model to have converged on a subtask once it gets mean test loss less than 0.1 bits on that subtask.

3. **Data scaling:** We observe that α_D is substantially higher than the theoretical prediction $\alpha/(\alpha + 1)$ for small α . We think this may be related to the fact that early-stopping cuts off training before all subtasks are learned as observed in Figure 2.7. In Figure 2.9, we show how the number of subtasks learned n , when we include subtasks learned after early-stopping, seems to be in line with theory: $n \propto D^{1/(\alpha+1)}$.

Better understanding the precise nature of power law scaling on multitask sparse parity is an interesting avenue for future work.

2.8.3 Additional results on language models

In Figure 2.11 we show how the distribution over losses changes across time during a training run, rather than across model scales like in Figure 2.3.

In Figure 2.13 we show additional examples from clusters discovered with QDG.

Details of application of QDG to LLMs

When applying QDG to language models, we use gradients within self-attention and MLP layers, but do not include embed, unembed, or layer norm gradients when we flatten and concatenate gradients into a vector g .³ We choose samples (x_i, y_i) for which our 19m-parameter model achieves a cross-entropy loss less than 0.1 nats. We filter based on this criteria since (1) we cannot cluster samples based on model mechanism if the model does not have such a mechanism for performing prediction correctly on those samples and (2) our intuition that samples with particularly low loss are more likely to be monogenic. We further exclude samples which can be solved via induction

³We exclude gradients for embed and unembed parameters because they are high dimensional and also because they may contain information more about the input and output rather than the computations the model performs internally. We exclude layer norm gradients because they appeared to contain less information about clusters in toy experiments.

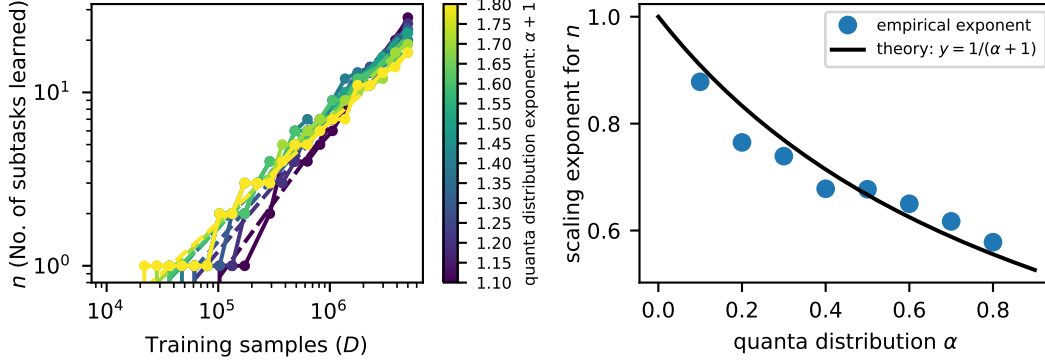


Figure 2.9: Number of subtasks learned (n), including subtasks learned after early-stopping would have terminated the training run, versus training samples D for a variety of α . We see that the relation $n \propto D^{1/(\alpha+1)}$ approximately holds, in line with theory. Deviation from theory for the scaling exponent of loss L w.r.t. D therefore likely originates from our failure to regularize network training, leading to early-stopping ending training before some subtasks can be learned.

on the context⁴, since such samples are quite common (possibly interfering with our task of finding diverse quanta) and since early experiments indicated that QDG had trouble clustering such samples together. We choose 10000 such samples to perform clustering on from the test set of The Pile. After computing the affinity matrix $\hat{C}$, we use the spectral clustering implementation from scikit-learn [45] with labels assigned via k-means.

2.8.4 Quanta discovery on TinyStories

We also apply QDG to TinyStories-33M, a language model trained on the TinyStories dataset [46]. We consider only tokens on which TinyStories-33M achieves a loss less than 1 bit of cross-entropy. We apply QDG to 10000 such samples, clustering their gradients with spectral clustering with `n_clusters = 400`. We show some samples from the resulting clusters in Figure 2.14. Many of these clusters reflect some simple recurring pattern in the TinyStories dataset, like predicting “time” after “Once upon a”, which many documents in the dataset start with. Some other clusters are more interesting however, like Cluster 11, which seems to involve predicting the correct noun in a sentence where that noun was referred to earlier in the sentence or in previous sentences.

2.8.5 The difficulty of estimating the power law exponent from clusters

In Section 2.5.1, when we looked at the distribution over elements in each cluster, we did not perfectly recover a Zipf distribution with exponent ≈ 1.08 that we expected from our theory. In this section, we describe the difficulty of accurately estimating such an exponent with our method.

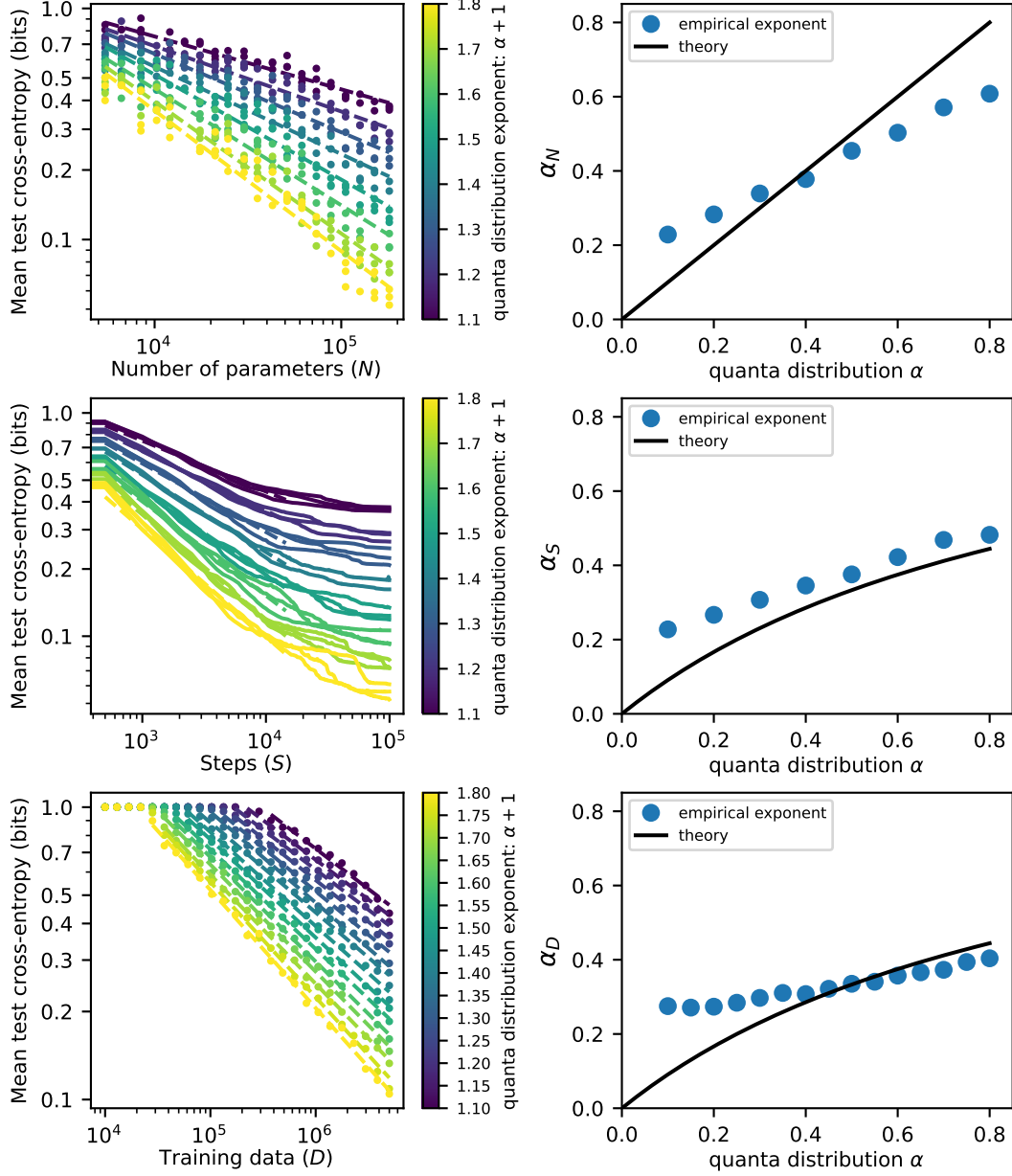


Figure 2.10: Scaling in parameters (N), single-epoch training time (S), and multi-epoch training samples (D) for varying quanta power law distribution parameter α on multitask sparse parity. We notice that scaling curves in steps S are typically steeper than the $\alpha_S = \alpha/(\alpha + 1)$ predicted from theory, and that for low α the scaling curves in D also deviate from theory substantially.

QDG on multitask sparse parity

As a first experiment, we performed QDG on multitask sparse parity, where there is a known, artificially-imposed power law distribution over subtasks. We train a width-500 single-hidden-

⁴We filter (copying) induction problems by excluding samples where the token which is to be predicted is the last token in a trigram which occurred earlier in the context. This is not a very comprehensive filtering scheme.

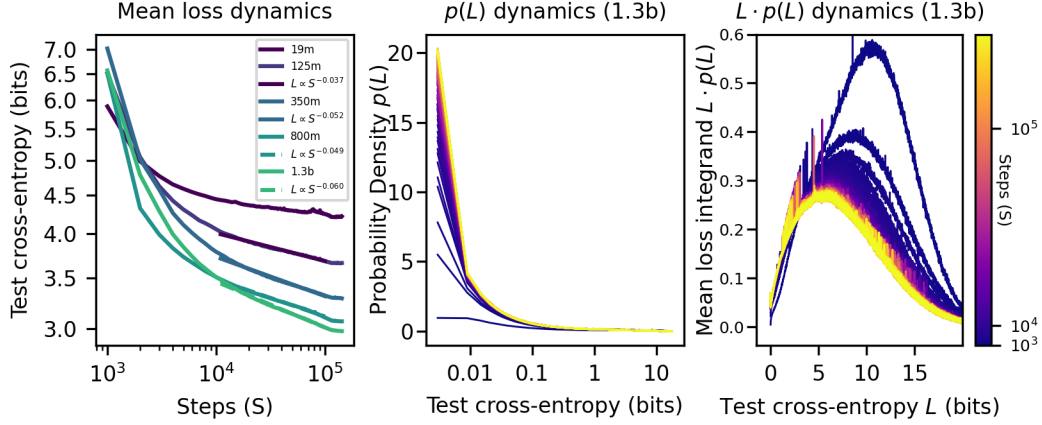


Figure 2.11: **Left:** Training curves (scaling w.r.t. steps S) of mean test loss for Pythia models. We measure exponents α_S between 0.037 and 0.06. **Center:** the distribution $p(L)$ over time. Over time, models achieve ≈ 0 loss on an increasing fraction of tokens, similar to scaling in model size. **Right:** The distribution $L \cdot p(L)$ over time.

layer ReLU MLP on multitask sparse parity with $\alpha = 0.4$ and with $n = 100$, $k = 3$, and $n_{\text{tasks}} = 500$. We then took 10000 samples which the network achieves ≈ 0 loss on (sampled from the Zipf distribution over subtasks with exponent 1.4). We compute gradients of cross-entropy loss w.r.t. all model parameters for these samples, and then perform QDG just like for LLMs. We show results in Figure 2.15. We plot the full similarity matrix where samples are ordered according to their a priori known subtask, rather than their cluster from QDG, and see a clear pattern where elements from the same subtask have on average higher angular similarity than elements between subtasks. However, from the rank-frequency plot of the clusters, we do not recover a slope of -1.4, but rather a lower slope of ≈ -1.1 . This shows that even when there is an exact decomposition of inputs into subtasks with a known Zipf distribution over these subtasks, that we do not perfectly recover this Zipf distribution from QDG.

A toy model of QDG uncertainty and bias

A toy model: To understand the bias of spectral clustering, we develop the following toy model. We assume the dataset has $N = 1000$ subtasks, each subtask containing $n_i = \lfloor \frac{A}{i^\alpha} \rfloor$ ($1 \leq i \leq N$) tokens ($A = 1000$). We use a Gaussian distribution $\mathcal{N}(\mathbf{m}_i, \sigma^2 \mathbf{I}_{d \times d})$ to model gradients within a subtask i , where d is the embedding dimension, σ is the noise level, and $\mathbf{m}_i$ is the Gaussian mean. $\mathbf{m}_i$ itself is drawn from the standard Gaussian distribution $\mathbf{m}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{d \times d})$. We define the similarity between two vectors $\mathbf{x}, \mathbf{y}$ to be $\text{sim} \equiv 1 + \frac{\mathbf{x}}{|\mathbf{x}|} \cdot \frac{\mathbf{y}}{|\mathbf{y}|}$. We compute pairwise similarity between all $\sum_{i=1}^N n_i$ tokens, and input the similarity matrix to the spectral clustering algorithm. We also need to specify the number of clusters k .

We have two hyperparameters in the toy model, the embedding dimension d and the noise level σ . We need to determine them such that this toy model can decently imitate LLM results (Figure 2.5). We fix $\alpha = 1$, sweeping $d = \{30, 100, 1000\}$, $\sigma = \{0, 0.5, 2.0\}$, and $k = \{100, 200, 500\}$. As shown in Figure 2.16, the high-dimension ($d = 1000$) large-noise ($\sigma = 2.0$) scheme seem to best agree with the LLM results, since the $k = 200$ curve can reproduce the sag and the cliff present in

LLM curves.

Estimating α from the frequency curve is hard, in fact, the slope depends on k and the region used to estimate it. However, we observe that different k curves form a clear envelope, whose slope is robust in a reasonably wide region. The envelope slope seems to indicate α . We fix $d = 1000$ and $\sigma = 2.0$, sweeping $\alpha = \{0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$. For each α , we estimate the slope of the envelope. Although there is clear correlation between the estimated envelope and α , if we use the envelope slope to estimate α , the error is on the order of 0.2, as shown in Figure 2.17.

2.8.6 Parameter and data scaling exponents across studies

In Figure 2.18, we show α_N and α_D (or possibly α_S , depending on the study) for a variety of prior studies of deep learning scaling, as compiled by Villalobos [47]. While the data is messy, it is intriguing that most of the Rosenfeld et al. [2] samples lie below the $\alpha_D = \alpha_N$ line, as our model would predict. The scaling exponents from Hoffmann et al. [7] are slightly closer to our prediction than the relation $\alpha_D = \alpha_N$, which has been proposed by other models of neural scaling laws. Overall though, the existing empirical results are too messy to definitively support or contradict our model.

2.8.7 Estimates of compute used for our experiments

Multitask sparse parity: Our training script takes roughly 1-4 hours (depending on network size) to perform a single-epoch training run on a GPU. When training multi-epoch on a fixed dataset, runs take typically between 3-10 minutes, with some outliers taking much longer. Our largest experiment was for Figure 2.10, where we trained networks of varying width on data with varying distributions over subtasks (with different power law exponents). 467 runs completed with a total running time of approximately 1450 hours. These experiments were run on a cluster with heterogeneous hardware. Available GPUs include NVIDIA A100, RTX A6000, QUADRO RTX6000, GEFORCE RTX2080TI, GEFORCE RTX2080, GEFORCE GTX1080TI, titan-x, and tesla-v100.

Pythia model scaling evaluations: We evaluated Pythia models on NVIDIA A100 80GBs. We do not have available the running time used when computing loss on approximately ten million tokens (for which we reported scaling statistics on), although it was likely less than an hour per model. The most expensive experiments were for Figure 2.11, where we evaluated the first four models in the Pythia suite across 143 checkpoints for a total of 572 evaluations. We likely used some hundreds of A100-hours for this, though possibly less than 100 hours.

QDG: We ran our QDG experiments on an NVIDIA A100 80GB. For the smallest Pythia model and for 10000 samples, it takes a few hours to compute the similarity matrix. We performed this computation only a handful of times.

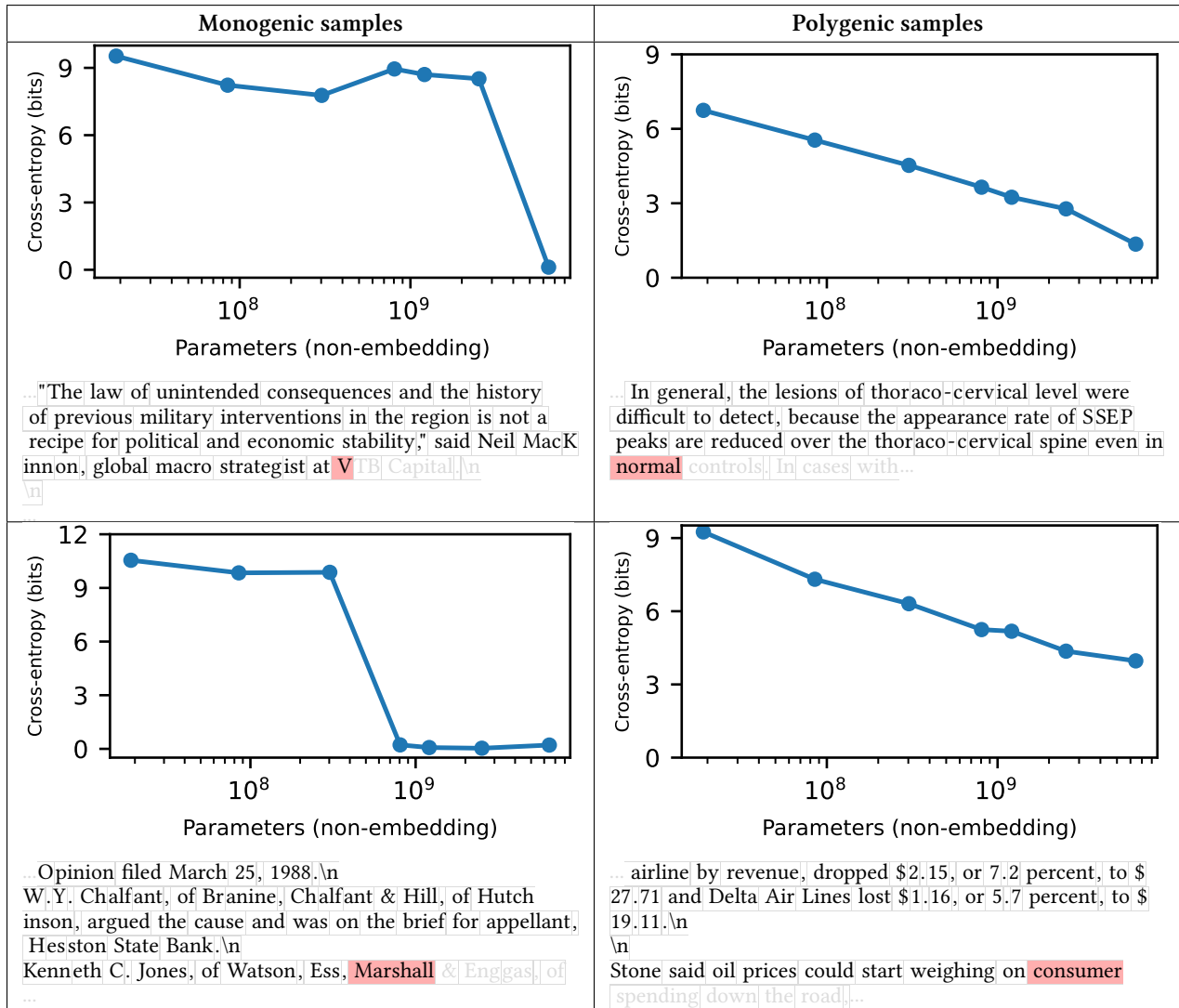


Figure 2.12: Additional LLM scaling curves on individual samples which exhibit sharp vs smooth improvement. If the Quantization Hypothesis is true for language modeling, then we would interpret samples with sharp drops as “monogenic” and samples with gradual progress as “polygenic”.

Cluster 146: comma after day of month	Cluster 269: “s” after starting year of decade
Sam Willard\n \n Samuel Steven Willard (born September 9 \n 215 U.S. 437 (1910)\n MECHANICAL APPLIANCE COMPANY\n v.\n CASTLEMAN.\n No. 48.\n Supreme Court of United States.\n Argued December 3, 1909.\n Decided January 3 Frederick W. Keator\n \n Frederick W. Keator (October 22, 1855 – January 31 United States Patent No. 6,073,124 (issued June 6, 2000) (“the ‘124 patent”). Microsoft in turn asserted counterclaims against NCI for infringement of three of its patents United States Patent Nos. 5,822,526, 5,999,914 and 5,794,006. Only terms of the ‘124 patent are presently before the Court; interpretation of claims in Microsoft’s patents will be interpreted in a separate Markman hearing to be held on November 15	Romford Ice Arena\n \n Romford Ice Arena was an ice rink located in Romford in the London Borough of Havering, England. The venue was built in the 1980s Although the novel continues to be the dominant medium of the crime-mystery-detective narrative, short stories by these contemporary authors may be found in numerous anthologies of the genre published during the 1990s Armed with the new Shenwei SW26010 chips, a new supercomputer has surged to the top of the TOP500 list: Sunway TaihuLight. It is nearly three times as fast as Tianhe-2, being benchmarked in Linpack as being able to perform 93 quadrillion calculations each second (93 petaFLOPS). To put this achievement in context, modern desktop PCs are already more powerful than the top ranked supercomputers from the early 1990s Early activism\n He began a lifetime involvement with revolutionary politics in the late 1930s Yesterday, visitor Greg Davidson commented that he was searching for songs played on the local forecast back in the late ‘80s In 1954, the couple published Living the Good Life which inspired many young, educated Americans to create simpler, rural lifestyles and the back-to-the-land movement of the 1960s
Cluster 278: colon after CSS property	Cluster 292: protocol separator in URLs
<pre>.rickshaw_graph.detail {\n pointer-events: none;\n position: absolute;\n top: 0;\n z-index: 2;\n background: rgba(0, 0, 0, 0.1);\n bottom: 0;\n width:</pre> <pre>@import './././assets/sass/spin';\n\n.app-header {\n background-color: #282c34;\n min-height: 100vh;\n display:</pre> <pre>See: http://jsfiddle.net/mWFGZ/1/\nhtml, body {\n margin: 0;\n padding:</pre>	<pre># \n\n# \n\n# For questions please refer to: \n\n# https://</pre> When I run the below code, I am getting an error: MongoError: The dollar (\$) prefixed field '\$push' in '\$push' is not valid for storage.\nI put this together based on the docs: https:// Gruber, Martin A. Views of the National Zoological Park in Washington, DC, showing Exhibit, 1919. Retrieved from the Digital Public Library of America, http:// But as citizens, our responsibility is to look beyond the anecdote. We journalists try to make sure that if a tree falls in the forest, it won't go unnoticed. Still, if we get so taken by the trees that we don't see the forest, we'll all be lost.\n Rex Smith is editor of the Times Union. Share your thoughts at http://

Figure 2.13: Additional examples of clusters of inputs discovered by QDG. Like in Figure 2.1, we used 10000 samples and `n_clusters` of 400.

Example “quanta” for the TinyStories dataset	
Cluster 11: predicting the correct noun	Cluster 31: “time” after “Once upon a”
Lily asked her mom, "Can I touch the sunflower?" Her mom replied, "No, Lily. The sun flower is not for touching. It's for looking at." \n Lily was sad, but she understood. The next day, Lily rode her bike past the sun Once upon a time, there was a big house with a door. The door was brown and it could move when people opened it. One day, a little boy came to the house and he saw the impressive door. He wanted to open it and see what was inside. So he moved the door When they got to the park, Tim saw a man with a sack. The man had found Tim's wagon and put it in the sack to keep it safe. Tim was happy to have his wagon back and thanked the man. He put his wagon Lily and Ben went to the park with Mom. They saw a big pond with many ducks and swans. Lily liked the swans. They were white and graceful. She wanted to feed them some bread.\n "Mom, can I give some bread to the sw Lily and Ben nod. They promise to be careful. They ask mom to read the letter to them. Mom smiles. She reads the letter. It is from grandma. She says she loves them a lot. She sends them kisses and hugs. Lily and Ben are happy. They send kisses and hugs back to grandma. They thank mom for the letter	Once upon a time Once upon a time Once upon a time Once upon a time Once upon a time Once upon a time Once upon a time Once upon a time
Cluster 75: comma after temporal phrase	Cluster 77: beginning of quote
One day, Jack wanted to show off his cool sunglasses. He spotted a nice patch of grass in the park, and he decided to sit down and enjoy the sun. He put his sunglasses on and just sat. He felt so special.\n A few moments later, Jack happily agreed and they started the game. At first, it was a bit tricky for both of them to get the squash to the other, but after a few tries, Jack was a pro. He laughed and cheered as he ran back and forth to get the squash. \n Mommy and Jack played until the sun started to go down. Then, Bob loves to go on adventures. He went for a walk along the beach, looking for fun and exciting things. All of a sudden, Linda was a little girl who loved wandering around in nature. She was eager to explore and find what she could. One day, Linda was wandering around in the woods when she spotted a big yellow flower. She was so excited that she ran over to it. She scooped the flower up and inspected it more closely. Soon, Next, her mom told Emma to wipe the floor clean. Emma grabbed a cloth and wiped the floor. When she was finished, it was as clean as a new penny.\n Finally,	Jack showed her the big blue ticket and said, "This is my ticket. I'm going to lay it down." \n The girl asked, "Where will you lay it down?" \n Jack answered, When she found her mom she said, "Mom, I have news!" Her mom said, "What is it, Jane?" Jane said, "I want to stir something up and make it more fun!" \n Her mom said, The twin jumped back in surprise. She had never heard a flower talk before! She asked the flower, "Who are you?" \n "My name is Pinky," said the flower. \n The twin was now even more surprised. She asked, He asked Jill, "What's in this jar?" Jill smiled and said, "It's sugar! Would you like some?" \n Jack shook his head and said, "No, thank you. I don't think I should eat sugar. My mom won't allow it." \n Jill nodded and said, Nearby, her mom was watching and called out, "Lucy, come here! What's that you have there?" \n Lucy proudly held up the hoop and announced,

Figure 2.14: Examples of clusters within the TinyStories dataset, discovered by QDG on the TinyStories-33M model. Here we just show samples from four out of 400 n_clusters

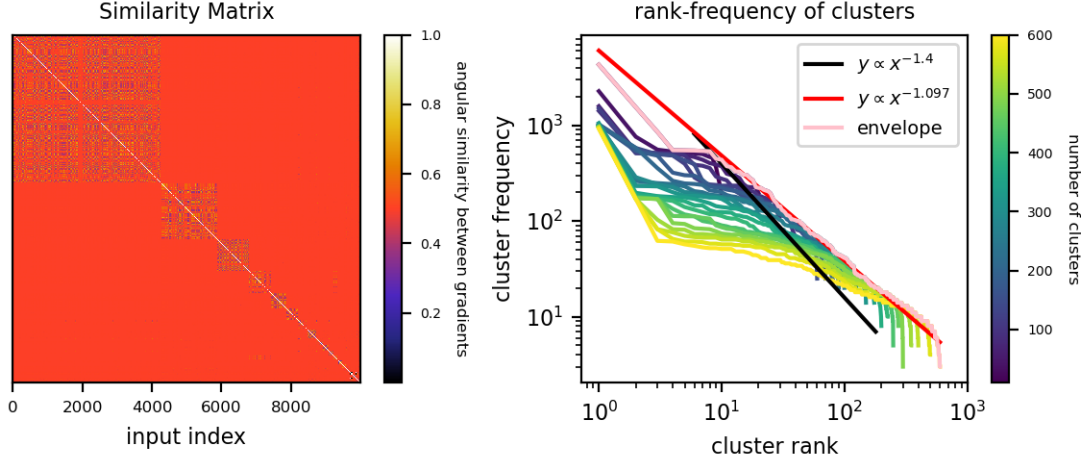


Figure 2.15: Similarity matrix and rank-frequency plots from QDG on multitask sparse parity. Despite sparse parity having a known decomposition into subtasks which are power law distributed in frequency, we do not recover this same power law from samples. We used $\alpha = 0.4$ for the frequency distribution for an expected rank-frequency power law exponent of -1.4, but measure a rank-frequency envelope slope closer to -1.1.

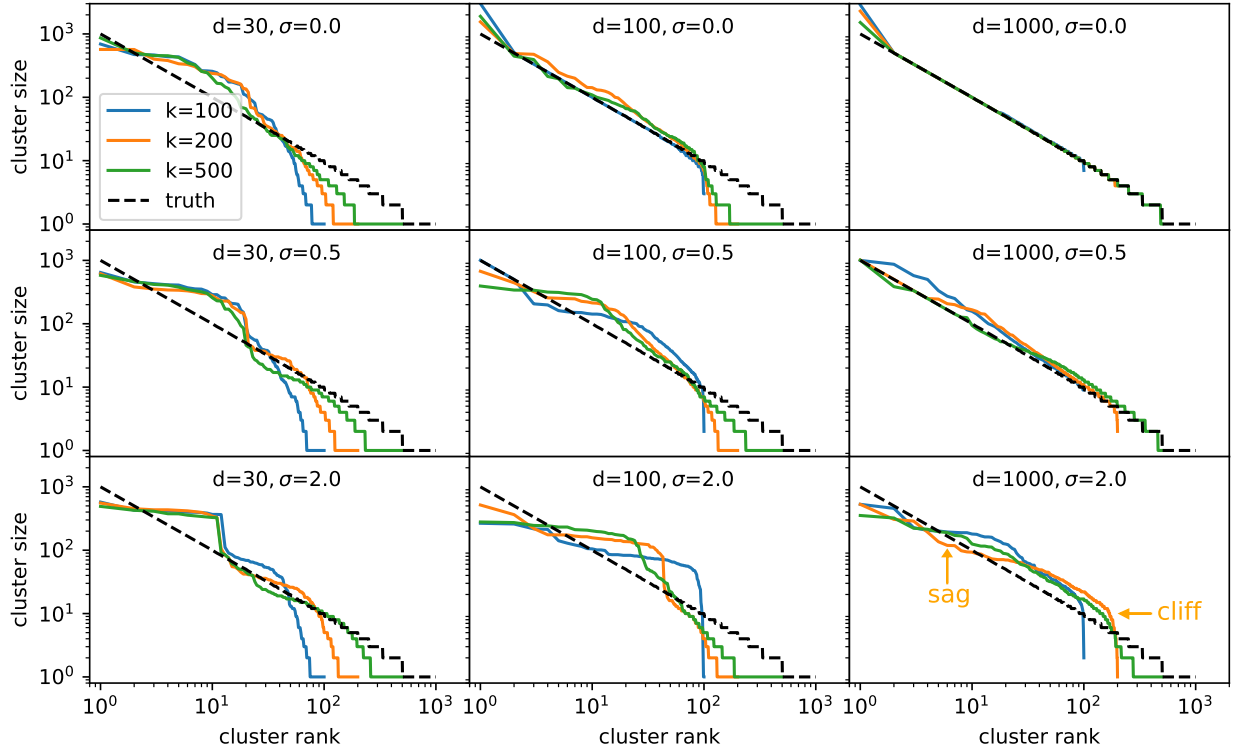


Figure 2.16: To understand the bias of spectral clustering, we apply spectral clustering to a toy model with different embedding dimension d , noise scale σ and number of cluster k . The high-dimension ($d = 1000$) large-noise ($\sigma = 2.0$) scheme seems to best agree with the LLM results (Figure 2.5).

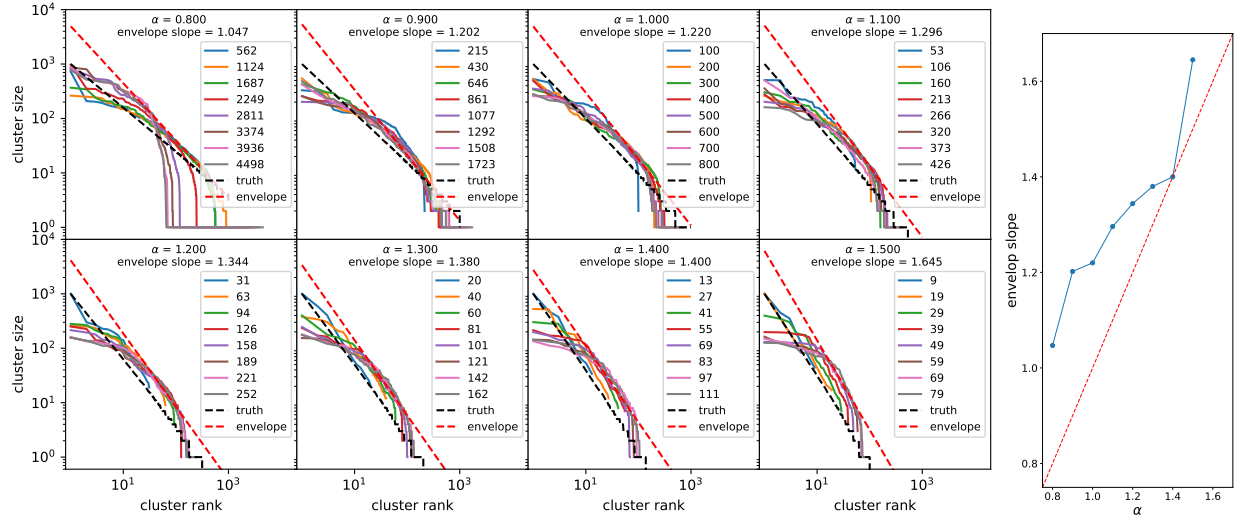


Figure 2.17: The difficulty of measuring α from curves. We apply spectral clustering to a toy model with different α and number of clusters k . For a fixed α , different k curves define an envelope. One could use the envelope slope to infer α , but this incurs errors around 0.2.

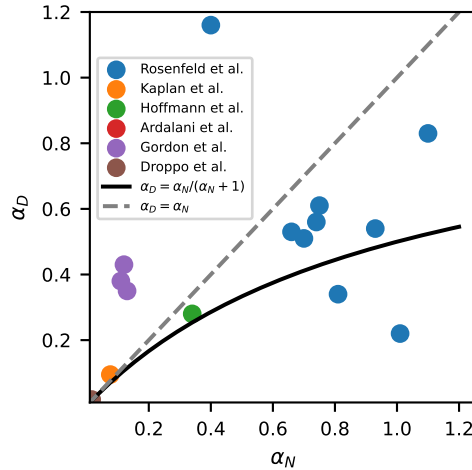


Figure 2.18: Parameter and data scaling exponents from various studies of deep learning scaling, compiled from the database of neural scaling laws from [47]. Our model of scaling predicts that $\alpha_D = \alpha_N / (\alpha_N + 1)$, indicated with the solid black line. Visible points are from [2,3,5,7,48]. [49] is above the visible window of the figure.

Chapter 3

On the Creation of Narrow AI: Hierarchy and Nonlocality of Neural Network Skills^{*}

In the previous chapter, we conjectured that the computation performed by neural networks like large language models in some manner decomposes into a large number of simpler modules, which each enable some separate skill. We assumed that these modules are *independent*, and that they are learned at a rate determined just by their “use frequency” in the data. But are these modules actually independent? Human skills, after all, are often taught in a curriculum, with more advanced skills only being learnable after more primitive ones are learned first. It may make sense then to imagine that our modules live in a kind of hierarchical dependency graph, and the learning process traverses the graph outwards, first learning primitive skills, then learning advanced skills which depend on them. In this case, the order in which skills are learned is not just determined by their “use frequency”, but also by the structure of this graph. In this paper, we study hierarchical structure between skills/modules. Most notably, we generalize the multitask sparse parity task to have hierarchical structure between the subtasks.

This work also begins to study how modules/skills are implemented within our networks. Crucially, we find that they are often not localized, by default, into a small subset of neurons. Instead, the computation is more distributed. However, with special training, we can move these modules into alignment with a small number of neurons, revealing their simplicity. This discussion emphasizes that the modularity we refer to in this thesis is not a naive one, where specific computations are localized to a small set of model components. Instead, the modularity is much more implicit, and takes special insight and effort to reveal.

^{*}Eric J. Michaud, Asher Parker-Sartori, and Max Tegmark. “On the creation of narrow AI: hierarchy and non-locality of neural network skills.” *Advances in Neural Information Processing Systems* 39 (to appear), 2025. Preprint: [arXiv:2505.15811](https://arxiv.org/abs/2505.15811).

Abstract

We study the problem of creating strong, yet narrow, AI systems. While recent AI progress has been driven by the training of large general-purpose foundation models, the creation of smaller models specialized for narrow domains could be valuable for both efficiency and safety. In this work, we explore two challenges involved in creating narrow AI systems, having to do with basic properties of how neural networks learn and structure their representations. The first challenge regards when it is possible to train narrow models from scratch. Through experiments on a synthetic task, we find that it is sometimes necessary to train networks on a wide distribution of data to learn certain narrow skills within that distribution. This effect arises when skills depend on each other hierarchically, and training on a broad distribution introduces a curriculum which substantially accelerates learning. The second challenge regards how to transfer particular skills from large general models into small specialized models. We find that model skills are often not perfectly localized to a particular set of prunable components. However, we find that methods based on pruning can still outperform distillation. We investigate the use of a regularization objective to align desired skills with prunable components while unlearning unnecessary skills.

3.1 Introduction

Today, the most competent AI systems in *any* particular domain are general systems that are relatively competent in *every* domain. The best models at math and coding are also broadly knowledgeable about a very diverse array of topics, from Roman history to home cooking recipes to medical diagnostics. And when domain-specific models are created today, they are typically general foundation models fine-tuned on a particular task, rather than new models trained from scratch [1,2], though with some notable exceptions [3]. This state of affairs is convenient and powerful, since a single general model can be used for a variety of applications [4].

However, generality has downsides for both efficiency and safety. For instance, AI systems used as coding assistants possess a large amount of knowledge which is never needed in those applications. Instead, we might like to use smaller, specialized networks which preserve the coding knowledge of general systems without the same breadth of irrelevant knowledge. Narrow systems may also pose fewer safety risks than general systems. For instance, narrow systems may have fewer dangerous capabilities that pose CBRN risks [5], be easier to understand mechanistically [6,7], or be easier to verify properties of [8–10]. More imaginatively, for systems to operate autonomously in the world requires a large breadth of skills, and an ecosystem of narrow “tool AI” systems may therefore reduce loss-of-control risks and better support human agency over the long term [11].

In this work, we investigate some basic questions about how neural networks *learn* and *implement* skills that are relevant to the problem of creating narrow AI systems. As summarized in Figure 3.1, we focus on two main themes. First, we consider the question of when it is possible to train well-performing neural networks from scratch on a narrow data distribution. Through experiments on a novel synthetic task with *hierarchical structure*, we find that it can be necessary to train networks on a broad distribution to efficiently learn narrow tasks within that distribution. These results contribute to a growing literature on how task structure influences neural network learning dynamics [12–15], but with special relevance to the problem of creating narrow AI. Second, we consider the question of whether one can use pruning to turn broad

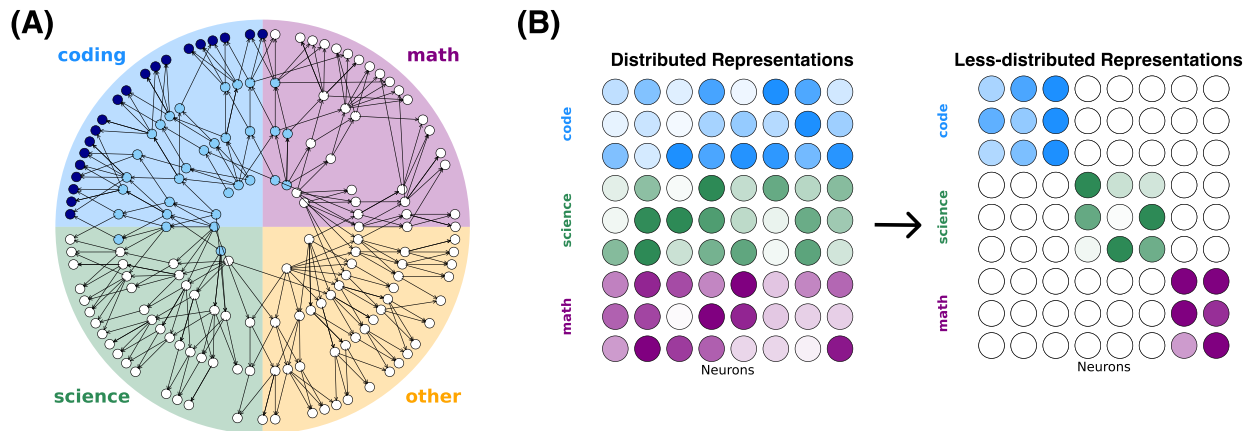


Figure 3.1: We study two challenges to making strong, narrow-purpose AI models. **(A)**: Data may have *hierarchical* structure. If skills have a hierarchical dependence, where some skills are only learnable after more primitive skills are learned first, then it sometimes may be necessary to train on a broad distribution of data to learn certain narrow skills within that distribution. These dynamics may mean that general-purpose models must be trained to achieve certain performance on some domains. **(B)**: Model features are *distributed*. By default, skills may not be localizable to a particular set of model components (e.g. neurons). In this case, pruning of model components won’t precisely retain wanted skills and remove unwanted skills from models. We explore methods for aligning the model features relevant to particular domains with a smaller subset of model components while unlearning others.

networks into smaller narrow ones. We find that the *nonlocality* of network representations to prunable model components poses a challenge for this goal. While distributed representations have been extensively studied in the context of neural network interpretability [16] and in the classical connectionist literature [17,18], we study how this property of neural network computation impacts pruning and unlearning [5] for creating narrow AI. Our specific contributions are as follows:

- We describe a synthetic task, *compositional multitask sparse parity* (CMSP), extending the multitask sparse parity task of Michaud et al. [19]. We find that networks trained on this task exhibit extremely strong curriculum learning effects, where it is necessary to train on a broad distribution of tasks in order to learn certain other tasks.
- We study pruning and unlearning on our networks trained on CMSP. We observe that tasks are often distributed and distinct subtasks are entangled, making pruning an imperfect strategy for “narrowing” the breadth of network skills. However, we find that a simple group-sparsity regularization objective can be used to sparsify networks and unlearn skills.
- We perform an empirical comparison of methods for creating narrow systems on MNIST and in LLMs. We tentatively find that methods based on pruning outperform distillation and training networks from scratch for the creation of smaller, more narrow systems.¹

¹Code for the experiments in this paper can be found at: <https://github.com/ejmichaud/narrow>.

This work is organized as follows. In Section 3.2, we briefly describe methods. In Section 3.3 we perform a detailed case study of training and pruning on the CMSP task. We define the task (Section 3.3.1), study network training dynamics on it (Section 3.3.2), find distributed representations in these networks (Section 3.3.3), and use a regularization method for pruning and unlearning in these networks (Section 3.3.4). We then compare various methods for creating narrow models, studying MNIST in Section 3.4 and language models in Section 3.5. We discuss related work in Section 3.6 and conclude in Section 3.7.

3.2 Methods

Pruning: We aim to preserve the performance of a model $f(\cdot; \theta)$ on a distribution $\mathcal{D}_N$ while pruning model components. Let g be a collection of parameter indices i corresponding to prunable components of the model (e.g. the in-weights and out-weights of a neuron), and let G denote the collection of all such groups. After ablating a specific g , we denote the new parameters θ_g^* . To perform pruning, for each $g \in G$, we compute an *ablation score* $s_g = |\mathbb{E}_{(x,y) \in \mathcal{D}_N} [L(f(x; \theta), y) - L(f(x; \theta_g^*), y)]|$, the absolute change in the model’s expected loss L after pruning g . We sort groups by their ablation score and prune greedily to the desired sparsity from lowest to highest ablation score. Where feasible, we estimate s_g empirically by manually ablating the group across many samples from $\mathcal{D}_N$. When this is computationally intractable, we instead use a linear estimate which we refer to as an *attribution score* after [20,21]: $\hat{s}_g = |\sum_{i \in g} \frac{\partial L}{\partial \theta_i}(-\theta_i)| = |\frac{\partial L}{\partial \theta} \cdot (\theta_g^* - \theta)|$, where $\partial L / \partial \theta$ is the model’s gradient on the distribution $\mathcal{D}_N$, which can be computed once and reused for all $g \in G$.

Regularization: We also experiment with making networks more prunable by performing additional training with a “group lasso” regularization penalty on the model weights [22–25]. The group lasso penalty R is the L1 norm of the L2 norms of each parameter group: $R(\theta) = \sum_{g \in G} \sqrt{\sum_{i \in g} \theta_i^2}$. This penalty incentivizes the weights to become sparse at the level of entire groups g . When we perform “group lasso training” on a distribution $\mathcal{D}_N$, we minimize the loss $\mathbb{E}_{(x,y) \sim \mathcal{D}_N} [L(f(x; \theta), y) + \lambda R(\theta)]$.

Distillation: We also explore distilling knowledge from a teacher model using the standard algorithm employed in [26], minimizing the KL divergence between student and teacher output distributions.

3.3 Case study: compositional multitask sparse parity

In this section, we conduct a detailed study of both curriculum learning and pruning on simple synthetic task, which we call *compositional multitask sparse parity* (CMSP).

3.3.1 Defining compositional multitask sparse parity (CMSP)

The compositional multitask sparse parity (CMSP) task is a simple extension of the *sparse parity* task recently studied in [27] and the *multitask sparse parity* task studied in [19], described below.

Barak et al. [27] studied neural network training dynamics on the *sparse parity* (SP) task. This is a binary classification problem on binary strings x of length n , where the label of a given sample

x is the parity of the bits at a subset I of k indices: $y = \bigoplus_{l=1}^k x[I_l]$. Strings x are sampled uniformly. Barak et al. observed that the loss curve for neural networks trained on this task exhibits a sharp drop after an initial plateau, a case of the sort of “emergence” which has been observed in LLMs [28,29] and in other toy settings [30].

Michaud et al. [19] extended the task to *multitask sparse parity* (MSP). In the MSP problem, input strings consist of $m + n$ bits, where the first m bits are called “control bits” and the last n bits are called “task bits”. These leading m control bits encode which “subtask” must be solved in each problem instance. Instead of a single length- k set of indices I , a collection of m sets of task bit indices (typically of equal length k) is chosen $\{I_1, \dots, I_m\}$. For each sample, only one control bit is ON (1) while the others are OFF (0). If control bit t is ON, then that sample’s label is the parity of bits I_t : $y = \bigoplus_{l=1}^k x[(I_t)_l]$. Michaud et al. [19] found that when subtasks are power law distributed in frequency, where the probability that control bit i is ON is $p_i \propto i^{-(\alpha+1)}$, the mean loss exhibits power-law scaling [31,32] while individual subtasks are learned at different times proportional to their frequency [33]. They conjecture that LLM learning dynamics may be similar.

One of the main limitations of the MSP task, and the associated model of neural scaling from [19], is that subtasks are independent. However, the world, and the problem of learning from it, intuitively has a hierarchical structure—in order to learn to do certain concepts and tasks, we must first learn simpler concepts and tasks. Accordingly, humans are taught in a curriculum, where simpler concepts and tasks are learned first, and then composed into more complex ones later. To capture this structure, we now introduce the toy task *compositional multitask sparse parity* (CMSP). We show some CMSP samples to the right:

CMSP is similar to MSP, except that (1) we require subtask indices to be disjoint: $I_i \cap I_j = \emptyset$ if $i \neq j$, and (2) multiple control bits can now be ON at the same time, in which case the label is the parity of the bits in the union of the indices for each subtask. If control bits i, j are both ON, the label is the parity of the task bits at indices $I_i \cup I_j$. We call samples for which only one control bit is ON “atomic” and samples for which multiple control bits are ON “composite”. So if $k = 3$ for all subsets I_i , then on atomic samples networks must compute the parity of 3 input bits, on composite samples with two control bits ON the label is the parity of 6 input bits, and so on. Above, we illustrated some CMSP data samples with $m = 3$, $n = 9$, and $k = 3$.

We call samples for which only one control bit is ON “atomic” and samples for which multiple control bits are ON “composite”. So if $k = 3$ for all subsets I_i , then on atomic samples networks must compute the parity of 3 input bits, on composite samples with two control bits ON the label is the parity of 6 input bits, and so on. Above, we illustrated some CMSP data samples with $m = 3$, $n = 9$, and $k = 3$.

CMSP Samples: $m=3, n=9, k=3$

		x	y
Atomic	1	100101001010	0
	1	100010100011	1
	0	10110111110	1
	0	01101101110	0
Compositional	1	110001101011	1
	1	110111001010	0
	0	11110001110	1
	1	111000110100	1
		Control Bits Task Bits	

3.3.2 Learning dynamics on CMSP

We find that neural network training on CMSP exhibits extremely strong curriculum learning effects. To denote different types of CMSP samples, we list the ON control bits for those samples. So, given a choice of m, n, k and $I_1, \dots, I_m$, $\mathcal{D}_{\{0\}}$ denotes all samples for atomic subtask 0, and $\mathcal{D}_{\{0,1,2,3\}}$ denotes all samples from the composite subtask where the first four control bits are ON.

For each subtask, there are 2^n possible samples in that subtask, since the m control bits are fixed but the n task bits are free.

We first train ReLU MLPs with 1-2 hidden layers of width 128 with the Adam optimizer on CMSP samples with $m = 4$, $n = 64$, $k = 4$, and 2000 samples per task (atomic or composite) per batch. We show the learning dynamics of these networks in Figure 3.2. We find that when we train on a dataset containing both atomic and composite samples $\mathcal{D}_{\{0\}} \cup \mathcal{D}_{\{1\}} \cup \mathcal{D}_{\{2\}} \cup \mathcal{D}_{\{3\}} \cup \mathcal{D}_{\{0,1,2,3\}}$ in equal proportion, atomic tasks are learned before composite ones.

Something interesting happens when we remove atomic samples from the dataset: we find that learning composite tasks takes dramatically longer. When we train on $\mathcal{D}_{\{0\}} \cup \mathcal{D}_{\{1\}} \cup \mathcal{D}_{\{2\}} \cup \mathcal{D}_{\{3\}} \cup \mathcal{D}_{\{0,1,2,3\}}$ with 10000 samples per batch (2000 per subtask), across 40 seeds, we find that 27/40 networks converge on composite subtask $\mathcal{D}_{\{0,1,2,3\}}$ within 2×10^9 samples, and the networks that do converge typically converge within 2×10^8 samples (bottom left). However, when we train just on $\mathcal{D}_{\{0,1,2,3\}}$, with a batch size of 2000 samples, we find that 0/40 networks converge within 2×10^9 samples in Figure 3.2 (bottom right). It is much more efficient to train on a broader distribution in order to learn subtask $\mathcal{D}_{\{0,1,2,3\}}$ than just training on $\mathcal{D}_{\{0,1,2,3\}}$ on its own.

This result makes sense given the exponential hardness of learning parities [34]. Since the atomic task bits are disjoint in CMSP, networks can compute the parity of the composite tasks by computing the parity of features learned for the atomic tasks, reducing the complexity of the

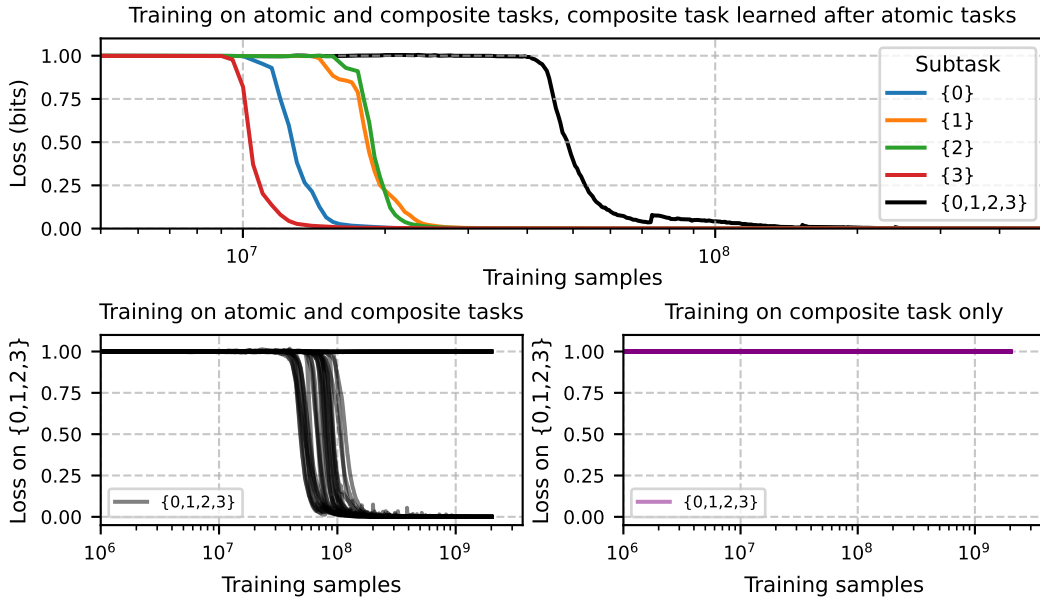


Figure 3.2: Training dynamics on compositional multitask sparse parity. **Top:** training dynamics for a single network trained on four atomic subtasks $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, and their composition $\{0,1,2,3\}$. **Bottom left:** loss on compositional subtask $\{0,1,2,3\}$, when training on atomic subtasks and their composition, as in the top subplot, across 40 network seeds (depth 3, width 128). **Bottom right:** loss on compositional subtask $\{0,1,2,3\}$, when only training on samples from that composite task, *without* also training on atomic tasks, across 40 network seeds (depth 3, width 128). We find that removing the atomic tasks prevents our networks from learning the composite task. We report the minimum loss within the previous 100 steps of training to filter out loss spikes.

problem. For instance, one could imagine that depth-3 networks would compute the parity of the atomic tasks in the first layer, then the composite tasks in the next layer. Curiously though, we find that single-hidden-layer (depth-2) networks can still take advantage of a curriculum in Section 3.8.1. We leave a mechanistic analysis of how networks learn these tasks to future work.

These sorts of dynamics may be a part of the explanation for why large-scale general-purpose models perform so strongly at many narrow, valuable tasks. We want to emphasize, however, that the toy task where we observe these curriculum effects is fairly contrived. It is not clear to what extent there are similarly strong effects on real-world tasks, and in many domains it is in fact possible to train narrow specialized models by only training on data in that domain. For instance, self-driving cars do not need to be trained on a broad corpus of text like LLMs.

Our work shows that for *some* types of tasks, it may be necessary to train on a broad data distribution in order to learn some subtasks efficiently. Accordingly, we now turn to the question of how to efficiently transfer knowledge from large models into smaller specialized ones.

3.3.3 Nonlocal representations in CMSP networks

If the circuitry for some subtasks were localized to a particular set of neurons, and the circuitry for other subtasks were localized to different neurons, then the task of specializing broad networks into narrow ones would be trivial. One could simply prune away neurons (or other model components, e.g. attention heads in transformers) associated with some subtasks, and keep others. However, often the situation seems more complicated than this ideal, which will be a focus for the rest of this paper.

A related problem has recently been studied in neural network interpretability, where it has been observed that individual computational units in neural networks, such as neurons, are *polysemantic*, activating across a wide variety of unrelated inputs [35,36]. Accordingly, many assume that the true model “features” do not align with architectural components like neurons. Multiple explanations have been proposed for this phenomenon. One is simply that the model architecture does not always “privilege” a particular basis [37,38], though other incidental reasons for polysemanticity have also been proposed [39]. Another explanation of polysemanticity is the *superposition hypothesis* [36,40]. The superposition hypothesis suggests that the need to represent more features than there are dimensions or neurons prevents features from being represented as orthogonal directions in the feature space, and therefore all features cannot be aligned with standalone model dimensions or neurons. Recently, studies that use sparse autoencoders to identify monosemantic model features have found that most features are highly distributed across a large number of dimensions of activation space [16].

In our CMSP networks, we observe a related problem. By default, without any explicit regularization, there is no incentive for the network to localize certain circuits cleanly into a particular set of neurons. We train networks with two hidden layers on a dataset of CMSP samples with $m = 6$, $n = 18$, $k = 3$ with two different skill trees: $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1,2\}$, and $\{3\}$, $\{4\}$, $\{5\}$, $\{3,4,5\}$ until convergence. In Figure 3.3, we visualize the connectivity of a 2-hidden-layer MLP trained on this dataset, and see that the network is densely connected without obvious structure.

We attempt to prune this network to retain performance on subtask $\{0,1,2\}$ while unlearning $\{3,4,5\}$. With MLPs, our groups of parameters g are the in-weights, bias, and out-weights for each hidden layer neuron, so that $|G|$ is the total number of hidden neurons. We compute ablation scores on the distribution $\mathcal{D}_N = \mathcal{D}_{\{0,1,2\}}$ (estimated on 2000 samples) and prune greedily, as described

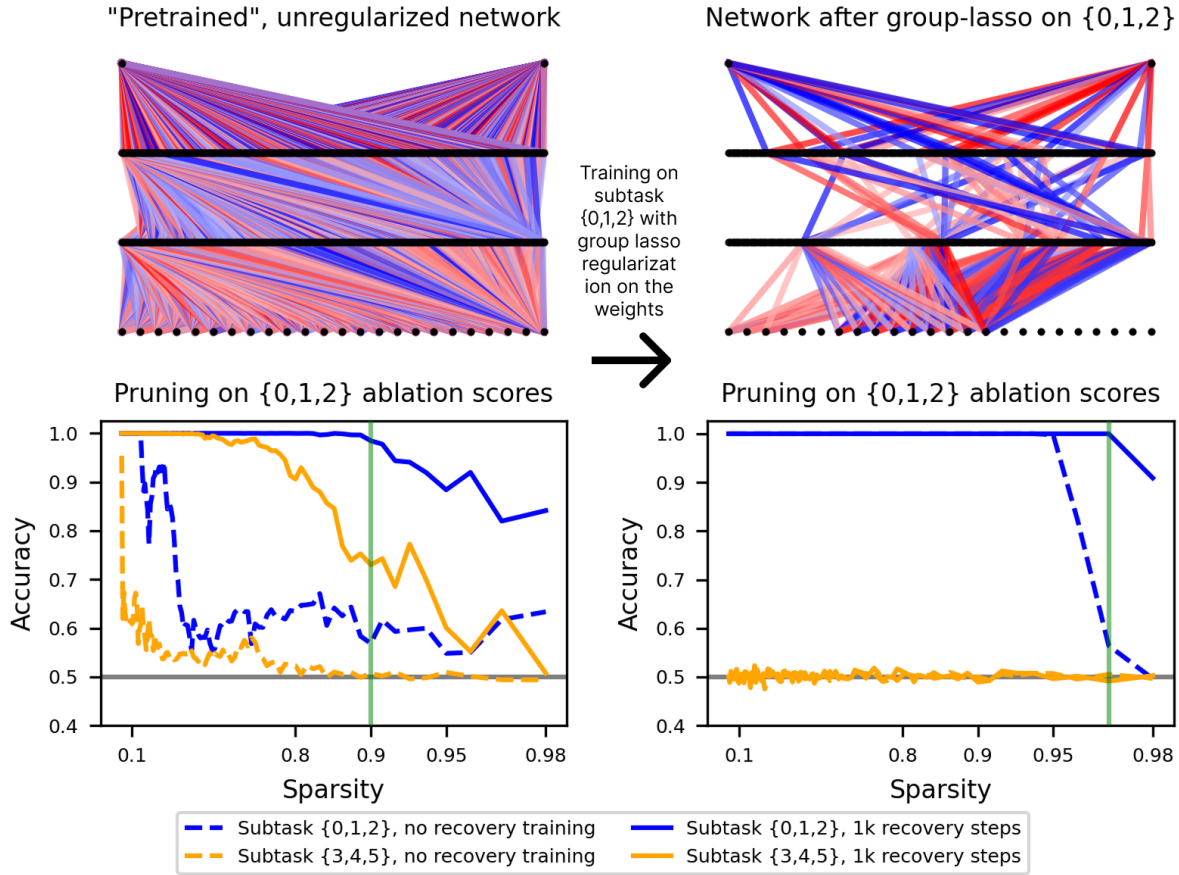


Figure 3.3: **Top:** We visualize the connectivity of 2-hidden-layer MLPs trained on a CMSP distribution with subtasks $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1,2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{3,4,5\}$, visualized before (**left**) and after (**right**) regularizing network weights with the group lasso sparsity penalty while training on subtask $\{0,1,2\}$. We find that network connectivity becomes sparse after regularizing. Negative weights are shown in blue and positive ones in red, with width proportional to the norm of the weight. **Bottom:** we show how pruning affects task performance on subtasks $\{0,1,2\}$ and $\{3,4,5\}$ at varying sparsity levels. We prune neurons based on the absolute change that ablating them has on the loss on subtask $\{0,1,2\}$. We find that subtasks here are nonlocal and entangled in the “pretrained” network (**left**). As we prune neurons according to their relevance on subtask $\{0,1,2\}$, at the sparsity at which performance on subtask $\{0,1,2\}$ accuracy drops below 98% (green line), we can still recover some performance on subtask $\{3,4,5\}$ with a small amount of additional training. Thus naive pruning here has not completely and robustly unlearned subtask $\{3,4,5\}$. However, after regularizing the weights (**right**), we find that not only we can more aggressively prune the network, but we have also robustly unlearned subtask $\{3,4,5\}$. Note that the degree to which subtasks are nonlocal and entangled in the pretrained networks depends on seed and width, and we show a variety of additional curves in Figure 3.10.

in Section 3.2. In Figure 3.3, we show accuracy on subtask $\{0,1,2\}$ vs. sparsity with this pruning strategy. When applied naively, network performance tends to degrade quickly, since without explicit regularization the network is not optimized to be naively prunable (see in Appendix

Figure 3.8 and Figure 3.9 for more on how ablation scores for each subtask vary across neurons). However, when we perform an additional 1000 steps of training on $\mathcal{D}_{\{0,1,2\}}$ (5000 samples per batch) after pruning (after either removing neurons from the architecture or pinning pruned weights at zero) we can recover performance at higher sparsity levels. Since we are training just on composite samples, 1000 steps is not enough to re-learn this task on its own, so if we can recover performance, that will be because the mechanisms for the task were somewhat preserved after pruning. We observe that often, the tasks $\{0,1,2\}$ and $\{3,4,5\}$ are *entangled* – when we prune as aggressively as we can while being able to recover performance on subtask $\{0,1,2\}$, we can often still recover some performance on subtask $\{3,4,5\}$. However, for our CMSP networks the degree of entanglement is highly seed-dependent, and sometimes the subtasks are disentangled enough that pruning as aggressively as possible on one subtask does in fact robustly unlearn the other. We show pruning curves across seeds and widths in Appendix Figure 3.10.

3.3.4 Regularizing to “narrow” networks

We find that we can use a simple regularization penalty to simultaneously unlearn some tasks while incentivizing the network to move its features to be less distributed, allowing for more aggressive pruning. As described in Section 3.2, we simply perform additional training on $\mathcal{D}_{\{0,1,2\}}$ with a group lasso sparsity penalty on the network weights. We aim for this penalty to “clean up” circuitry [41] not relevant to prediction on $\mathcal{D}_{\{0,1,2\}}$ while also sparsifying the weights across groups of parameters, allowing for easier pruning.

We apply this regularization to our CMSP networks, training with penalty strength $\lambda = 10^{-3}$ with a batch size of 2000 for 10000 steps. With this regularized network, we then re-compute neuron ablation scores and prune. In Figure 3.3 we find that, in CMSP networks, this method is effective at unlearning skills and making the network more prunable. We find that we can prune more aggressively while retaining performance on subtask $\{0,1,2\}$, and we are not able to recover performance on $\{3,4,5\}$ at any sparsity level. These results complete our case study on CMSP: in order to efficiently learn compositional tasks, we train on a broad distribution of tasks. Then, we can regularize the network to turn this broad network into a much smaller one which nevertheless retains the broad network’s performance on the target compositional task.

3.4 Pruning vs distillation: MNIST

We now consider the problem of creating narrow systems in more natural domains, first on MNIST. As our narrow subtask, we choose only even digits from the original MNIST dataset [42]. We compare the resources required to achieve good performance on this narrow task when (1) training from scratch on the narrow task, and (2) distilling models from a general teacher on this task, (3) using group-lasso regularization to prune a large general model and (4) using attribution-based pruning and then recovery training on a large general model. Our teacher model is a ReLU MLP with two hidden layers each of width 1200, as in [26], and achieves 98.7% accuracy on the test set. When pruning, we use this same teacher model as our initial model and prune its hidden neurons to create a smaller network.

When using distillation (2), we use the approach of Hinton, Vinyals, and Dean [26] with $T = 20$. When pruning with the group lasso penalty (3), we use λ values ranging from 0.001 to 0.008.

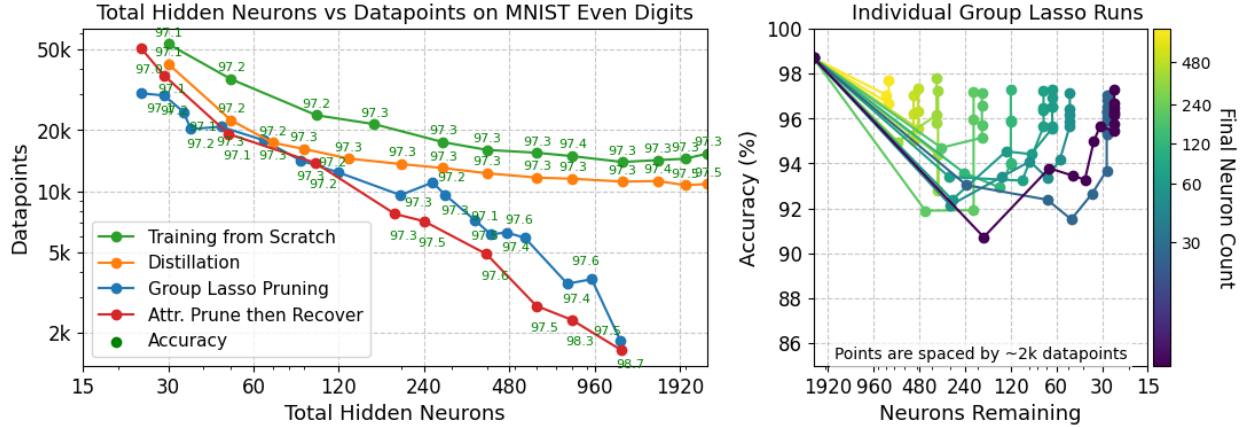


Figure 3.4: **Left:** We compare the performance of distillation, training from scratch, and two pruning approaches for creating small networks that classify MNIST even digits. Pruning-based approaches Pareto-dominate distillation, achieving high compression ratios with fewer datapoints. All points are averaged over 10 individual training runs. **Right:** When pruning using group lasso, it often helps to first prune rapidly, degrading performance, and then recover performance with no regularization. Each line represents a single training run, with a new point logged every 2,000 datapoints. Darker lines correspond to more aggressive pruning runs that attain lower final neuron counts.

Unlike in Section 3.3.4, we regularize and prune simultaneously, pruning neurons when their L2 norm drops below 0.05. When the number of remaining neurons drops below a target threshold, we remove the pruning penalty and continue training to recover lost performance during pruning. When we prune up front and then separately recover performance (4), we use attribution scores as described in Section 3.2.

To compare methods, we require that each method reach a test-set accuracy of 97 percent, and we then plot the frontier of neuron count versus datapoints subject to that threshold. As seen in Figure 4 (right), it is often optimal to first aggressively prune the network down to the desired size and further train it until it reaches the requisite accuracy.

We find that while group lasso pruning is highly sensitive to choices in hyperparameters, both pruning methods Pareto-dominate distillation and training from scratch, especially at high neuron counts. Moreover, while other methods cannot consistently bridge the 97 percent accuracy threshold with fewer than 25 hidden neurons, aggressive pruning can consistently shrink the network’s size to a lower absolute limit.

3.5 Pruning vs distillation: LLMs on Python documents

We next study LLMs. As our narrow task $\mathcal{D}_N$, we choose next-token prediction on Python documents in the [GitHub Code Dataset](#).

We first prune the neurons in the MLP blocks of Llama-3.2-1B [43]. Later, we will also consider pruning residual stream dimensions, which involves pruning all model parameters that “read to” or “write from” a dimension of the residual stream [37]. We use attribution scores when

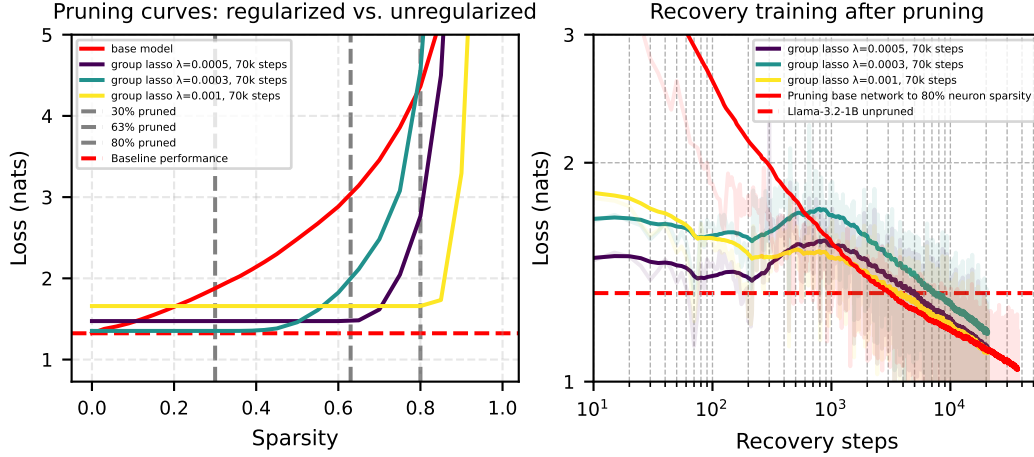


Figure 3.5: **Left:** Neuron sparsity vs. loss curve for LLMs finetuned with group lasso regularization with varying λ for 70k steps, as compared to the base network. Regularization flattens the sparsity vs. loss curve, at the cost of slightly degrading model performance. **Right:** After pruning our networks to 30%, 63%, and 80% sparsity for our runs with λ of $5e-4$, $3e-4$, and $1e-3$, respectively, we recover performance with additional training. We find that we can recover performance lost during pruning, including in the network that was pruned without first using group-lasso regularization training. The plot shows an exponential moving average of batch losses, with individual batch losses shaded in back.

pruning, and show that the attribution scores correlate moderately with true ablation scores in Appendix Figure 3.11. In Figure 3.5 (left), we show neuron sparsity vs loss curves. When pruning neurons naively, we see that loss increases quickly at low sparsity levels. We also experiment with applying group lasso regularization while further training on Python documents (learning rate of $2e-6$, max length 512, 18 documents per batch) and find that this training does indeed level out the sparsity vs. loss curve, albeit at a slight cost to the loss. Fortunately, we find that we can recover lost performance after pruning by doing a small amount of additional training on Python documents in Figure 3.5 (right). We find that despite the loss increasing substantially after naively pruning, we can also quickly recover that lost performance, and overall this strategy seems to be more efficient than using group lasso training. We therefore next compare naive pruning + recovery training against distillation and training networks from scratch.

We train networks with the Llama 3 [43] architecture of varying shape and size. We use a learning rate of $5e-4$, sequence length of 1024, and batch size of 64. For distillation, we use Llama-3.1-8B as a teacher with $T = 2$. For pruning + recovery, we prune Llama-3.2-1B to varying levels of neuron and residual stream sparsity, shown in Appendix Table 3.3. In Appendix Figure 3.12 we show learning curves for these three approaches. In Figure 3.6, we tentatively find that pruning substantially outperforms training from scratch and distilling a model from scratch on the data-parameter frontier. Given a target narrow network size and a fixed data budget, if one already has access to a general model, it appears to be more efficient to prune that model than it is to perform distillation.

We also experiment with pruning random model components instead of components selected by attribution score and find that random pruning performs similarly [44]! In Appendix Figure 3.13, we find that after a moderate number of recovery steps, attribution pruning and random pruning result in the same recovered performance. This result is in line with our earlier discussion of nonlocality, and the empirical findings of Bricken et al. [16], that monosemantic features are distributed widely across model components. While some studies have found some geometric similarity between functionally similar features [45,46], in our case it does not seem like the relevant features for our task are localized into a set of “Python” vs. “non-Python” neurons, at least that attribution pruning identifies.

Unlearning: We lastly evaluate whether neuron pruning can robustly unlearn downstream tasks. We consider three pruning methods applied to Llama-3.2-1B at sparsity levels of 30%, 63%, and 80%: (i) random neuron pruning, (ii) attribution-based pruning on Python code, and (iii) models trained with group-lasso regularization on Python documents from Figure 3.5. We evaluate each pruned model on four benchmarks: counterfact [47], AI2-ARC (ARC-Easy) [48], WMDP-Bio, and WMDP-Cyber [49] for hazardous knowledge. We found that the base model performed poorly on WMDP-Chem. For each model, we first measure baseline accuracy after pruning, then fine-tune on each benchmark’s training set with batch size 4, sweeping learning rates between $1e-6$ and $1e-4$ and reporting the best-performing configuration on the test set. In Table 3.1, we find that all pruning techniques, including random pruning, robustly unlearn non-Python skills with roughly similar effectiveness, except that at low sparsity, attribution pruning does not achieve much unlearning. This result contrasts somewhat with our findings on CMSP networks in Sections 3.3.3 and 3.3.4, where training with the group lasso penalty led to stronger unlearning than naive pruning.

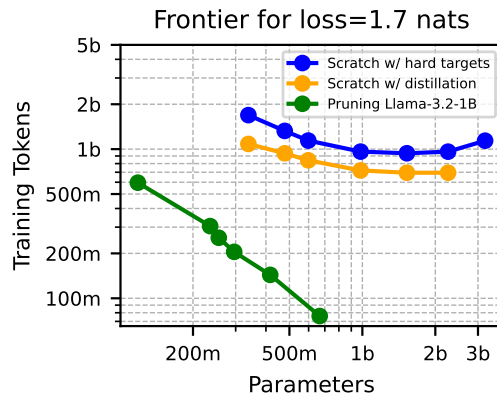


Figure 3.6: Frontier of network parameters vs. data required to achieve a certain cross-entropy loss on Python documents. For the task of creating a LLM specialized on Python documents, we find that pruning Llama-3.2-1B and then performing recovery training is much more efficient than training LLMs from scratch or distilling LLMs from scratch on the soft targets of Llama-3.2-8B.

Table 3.1: **Unlearning**: We prune Llama-3.2-1B using various techniques to create narrow Python-specific models, then evaluate on other tasks to assess whether pruning led to unlearning. Accuracies are reported for the pruned model before finetuning and after finetuning on a train set of each benchmark (non-FT→FT). ✓ indicates successful unlearning (FT score $\geq 10\%$ lower than base).

Method	Spar.	counterfact	AI2-ARC	WMDP Bio	WMDP Cyber
Base	0%	0.18→ 0.49	0.65→ 0.67	0.52→ 0.62	0.35→ 0.59
Random	30%	0.00→ 0.50	0.24→ 0.25 ✓	0.22→ 0.27 ✓	0.27→ 0.27 ✓
Attribution	30%	0.12→ 0.65	0.35→ 0.69	0.36→ 0.56	0.27→ 0.55
Group Lasso	30%	0.06→ 0.47	0.25→ 0.25 ✓	0.24→ 0.31 ✓	0.26→ 0.27 ✓
Random	63%	0.00→ 0.45	0.25→ 0.25 ✓	0.23→ 0.27 ✓	0.26→ 0.28 ✓
Attribution	63%	0.02→ 0.52	0.25→ 0.30 ✓	0.23→ 0.34 ✓	0.26→ 0.32 ✓
Group Lasso	63%	0.03→ 0.45	0.25→ 0.25 ✓	0.24→ 0.29 ✓	0.27→ 0.30 ✓
Random	80%	0.00→ 0.42	0.26→ 0.25 ✓	0.26→ 0.28 ✓	0.23→ 0.26 ✓
Attribution	80%	0.00→ 0.47	0.25→ 0.25 ✓	0.23→ 0.27 ✓	0.26→ 0.28 ✓
Group Lasso	80%	0.02→ 0.44	0.25→ 0.25 ✓	0.24→ 0.29 ✓	0.26→ 0.29 ✓

3.6 Related Work

Distillation. Many works have built on the original distillation work of Hinton, Vinyals, and Dean [26], seeking to transfer intermediate representations [50,51] and applying these techniques to language models [52,53], often after pruning [54,55]. Relevant to our discussion, Turc et al. [56] showed that pretraining the student before distillation can substantially improve results.

Pruning. Even early approaches to pruning used second-order methods for pruning weights [57,58], whereas our “attribution” scores are first-order. When training vision models, Zhou, Alvarez, and Porikli [25] and Wen et al. [24] used a group lasso penalty like we do, albeit while training on the full data distribution. Sanh, Wolf, and Rush [59] proposed “movement pruning” to prune weights during transfer learning. A variety of works have applied structured pruning to LLMs [60,61], including Xia, Zhong, and Chen [62] who develop a method for task-specific pruning. Vural and Erdogdu [63] provide evidence that pruning boosts sample efficiency by directing gradient updates to a sparse set of important directions.

Task Structure and Learning Dynamics. Several works have lately investigated the relationship between task structure and learning dynamics [12–15]. Liu et al. [64] also briefly study a task similar to CMSP to show that hierarchical relationships between tasks cause what they call “domino” learning dynamics. We also highlight the “Skill it!” paper of Chen et al. [65], who empirically investigate hierarchical learning dependencies for real LLM tasks.

Machine Unlearning. Many approaches to unlearning have been proposed [66–70]. Guo et al. [71] study how applying unlearning fine-tuning to different model components affects unlearning success, inspired by a mechanistic understanding of how knowledge is retrieved in LLMs. Lee et al. [72] show that distillation can be applied after unlearning fine-tuning to more robustly remove unwanted capabilities. Highly relevant is the work of Cloud et al. [73], who apply “gradient routing” during training to localize capabilities to different model components, allowing pruning to be used for unlearning.

3.7 Discussion

Limitations: This work explored some aspects of neural network learning which are relevant to the creation of narrow AI systems. Much work remains to be done in determining how relevant our findings are to real-world networks in various domains. We first emphasize that CMSP is a toy task, an existence proof that curriculum learning effects can be quite strong, but it is still unclear whether these dynamics occur on more natural datasets. For our pruning experiments, we emphasize that our greedy attribution pruning approach is quite simple, and we cannot rule out that more sophisticated pruning strategies would be more successful in preserving some skills while unlearning others [74]. Also, we did not optimally scale hyperparameters in our LLM experiments, so the performance of each of our models in Section 3.5 is likely imperfect. While we found that pruning a pre-existing Llama model was more efficient than training a model from scratch on Python documents, this could be due to our imperfect training setup, and not because one needs to train on a broad distribution of data to learn Python documents due to curriculum effects.

Conclusion: In this work, we have studied some potential challenges involved in creating narrow AI systems, having to do both with the structure of data and the structures learned internally by neural networks. Underlying this work is a perspective from *mechanistic interpretability*, that neural networks compute a variety of sparse *features* [35], each with a distributed representation [16,36,46], and that these features are computed from each other hierarchically in *circuits* [75–77]. First, we found that in order to learn certain complex features, we may have to first train on a broad set of samples which encourage the learning of simpler features. Second, because features are distributed across model components, it is a nontrivial problem to move a set of task-specific features and circuits into a smaller network, especially while *unlearning* unneeded circuits.

While a neural network’s computation across the whole data distribution may be quite complex, we hope that the computation that networks perform on any particular task will be reducible to something less complex. That less complex computation, whatever it is, might be interpretable as a circuit [76,77], or reduce to a simple program [78], or, as we studied here, could be instantiated in a much smaller network. However, this hope, and the question of whether it is possible to create narrow-purpose versions of today’s models, probes at a basic question about the nature of the intelligence of these models. If their apparent generality indeed results from their having learned a large, diverse set of crystallized, task-specific circuits, then we ought in principle to be able to create competent, specialized versions of these models by just transferring the relevant task-specific circuits. However, if the intelligence of our models, and intelligence more generally, is better understood as resulting from a single unified algorithm, then the basic prospect of creating narrow AI systems that are as strong as truly general ones could be a challenge.

Acknowledgements

We thank Ziming Liu, Josh Engels, David D. Baek, Tom McGrath, and Jamie Simon for helpful conversations and feedback. E.J.M. was supported by the NSF via the Graduate Research Fellowship Program (Grant No. 2141064) and under Cooperative Agreement PHY-2019786 (IAIFI).

References

- [1] Z. Ren, Z. Shao, J. Song, H. Xin, H. Wang, W. Zhao, L. Zhang, Z. Fu, Q. Zhu, D. Yang, et al. “DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition.” *arXiv preprint arXiv:2504.21801*, 2025.
- [2] P. Kassianik, B. Saglam, A. Chen, B. Nelson, A. Vellore, M. Aufiero, F. Burch, D. Kedia, A. Zohary, S. Weerawardhena, et al. “Llama-3.1-FoundationAI-SecurityLLM-Base-8B Technical Report.” *arXiv preprint arXiv:2504.21039*, 2025.
- [3] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, et al. “Qwen2.5-coder technical report.” *arXiv preprint arXiv:2409.12186*, 2024.
- [4] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al. “On the opportunities and risks of foundation models.” *arXiv preprint arXiv:2108.07258*, 2021.
- [5] F. Barez, T. Fu, A. Prabhu, S. Casper, A. Sanyal, A. Bibi, A. O’Gara, R. Kirk, B. Bucknall, T. Fist, et al. “Open problems in machine unlearning for ai safety.” *arXiv preprint arXiv:2501.04952*, 2025.
- [6] L. Bereska and E. Gavves. “Mechanistic Interpretability for AI Safety—A Review.” *arXiv preprint arXiv:2404.14082*, 2024.
- [7] L. Sharkey, B. Chughtai, J. Batson, J. Lindsey, J. Wu, L. Bushnaq, N. Goldowsky-Dill, S. Heimersheim, A. Ortega, J. Bloom, et al. “Open Problems in Mechanistic Interpretability.” *arXiv preprint arXiv:2501.16496*, 2025.
- [8] M. Tegmark and S. Omohundro. “Provably safe systems: the only path to controllable AGI.” *arXiv preprint arXiv:2309.01933*, 2023.
- [9] D. Dalrymple, J. Skalse, Y. Bengio, S. Russell, M. Tegmark, S. Seshia, S. Omohundro, C. Szegedy, B. Goldhaber, N. Ammann, et al. “Towards guaranteed safe ai: A framework for ensuring robust and reliable ai systems.” *arXiv preprint arXiv:2405.06624*, 2024.
- [10] M. Tegmark, S. Mindermann, V. Wilfred, and W. S. Lee. *The Singapore Consensus on Global AI Safety Research Priorities*. Conference Report. Singapore: 2025 Singapore Conference on AI: International Scientific Exchange on AI Safety, Apr. 2025. URL: <https://file.go.gov.sg/sg-consensus-ai-safety.pdf>.
- [11] K. E. Drexler. *Reframing Superintelligence: Comprehensive AI Services as General Intelligence*. Future of Humanity Institute Technical Report. 2019. URL: https://www.fhi.ox.ac.uk/wp-content/uploads/Reframing_Superintelligence_FHI-TR-2019-1.1-1.pdf.
- [12] R. Ramesh, E. S. Lubana, M. Khona, R. P. Dick, and H. Tanaka. “Compositional capabilities of autoregressive transformers: A study on synthetic, interpretable tasks.” *arXiv preprint arXiv:2311.12997*, 2023.
- [13] M. Okawa, E. S. Lubana, R. Dick, and H. Tanaka. “Compositional abilities emerge multiplicatively: Exploring diffusion models on a synthetic task.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 50173–50195.

- [14] C. F. Park, M. Okawa, A. Lee, E. S. Lubana, and H. Tanaka. “Emergence of hidden capabilities: Exploring learning dynamics in concept space.” *Advances in Neural Information Processing Systems*, 37, 2024, pp. 84698–84729.
- [15] E. Abbe, E. B. Adsera, and T. Misiakiewicz. “Sgd learning on neural networks: leap complexity and saddle-to-saddle dynamics.” In: *The Thirty Sixth Annual Conference on Learning Theory*. PMLR. 2023, pp. 2552–2623.
- [16] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [17] P. Smolensky. “Tensor product variable binding and the representation of symbolic structures in connectionist systems.” *Artificial intelligence*, 46(1-2), 1990, pp. 159–216.
- [18] G. E. Hinton. “Learning distributed representations of concepts.” In: *Proceedings of the Annual Meeting of the Cognitive Science Society*. Vol. 8. 1986.
- [19] E. Michaud, Z. Liu, U. Girit, and M. Tegmark. “The quantization model of neural scaling.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 28699–28722.
- [20] N. Nanda. *Attribution patching: Activation patching at industrial scale*. 2023. URL: <https://www.neelnanda.io/mechanistic-interpretability/attribution-patching>.
- [21] A. Syed, C. Rager, and A. Conmy. “Attribution patching outperforms automated circuit discovery.” *arXiv preprint arXiv:2310.10348*, 2023.
- [22] R. Tibshirani. “Regression shrinkage and selection via the lasso.” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1), 1996, pp. 267–288.
- [23] M. Yuan and Y. Lin. “Model selection and estimation in regression with grouped variables.” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 68(1), 2006, pp. 49–67.
- [24] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. “Learning structured sparsity in deep neural networks.” *Advances in neural information processing systems*, 29, 2016.
- [25] H. Zhou, J. M. Alvarez, and F. Porikli. “Less is more: Towards compact cnns.” In: *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer. 2016, pp. 662–677.
- [26] G. Hinton, O. Vinyals, and J. Dean. “Distilling the knowledge in a neural network.” *arXiv preprint arXiv:1503.02531*, 2015.
- [27] B. Barak, B. Edelman, S. Goel, S. Kakade, E. Malach, and C. Zhang. “Hidden progress in deep learning: Sgd learns parities near the computational limit.” *Advances in Neural Information Processing Systems*, 35, 2022, pp. 21750–21764.
- [28] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, et al. “Emergent abilities of large language models.” *arXiv preprint arXiv:2206.07682*, 2022.
- [29] C. Olsson et al. “In-context Learning and Induction Heads.” *Transformer Circuits Thread*, 2022. <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [30] N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.

- [31] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. “Deep learning scaling is predictable, empirically.” *arXiv preprint arXiv:1712.00409*, 2017.
- [32] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling laws for neural language models.” *arXiv preprint arXiv:2001.08361*, 2020.
- [33] N. Fonseca, S. H. Lee, C. Mingard, A. Louis, et al. “An exactly solvable model for emergence and scaling laws in the multitask sparse parity problem.” *Advances in Neural Information Processing Systems*, 37, 2024, pp. 39632–39693.
- [34] I. Shoshani and O. Shamir. “Hardness of Learning Fixed Parities with Neural Networks.” *arXiv preprint arXiv:2501.00817*, 2025.
- [35] C. Olah, A. Mordvintsev, and L. Schubert. “Feature Visualization.” *Distill*, 2017. <https://distill.pub/2017/feature-visualization>. DOI: 10.23915/distill.00007.
- [36] N. Elhage et al. “Toy Models of Superposition.” *Transformer Circuits Thread*, 2022. https://transformer-circuits.pub/2022/toy_model/index.html.
- [37] N. Elhage et al. “A Mathematical Framework for Transformer Circuits.” *Transformer Circuits Thread*, 2021. <https://transformer-circuits.pub/2021/framework/index.html>.
- [38] N. Elhage, R. Lasenby, and C. Olah. “Privileged bases in the transformer residual stream.” *Transformer Circuits Thread*, 2023, p. 24.
- [39] V. Lecomte, K. Thaman, R. Schaeffer, N. Bashkansky, T. Chow, and S. Koyejo. “What causes polysemanticity? an alternative origin story of mixed selectivity from incidental causes.” *arXiv preprint arXiv:2312.03096*, 2023.
- [40] S. Arora, Y. Li, Y. Liang, T. Ma, and A. Risteski. “Linear algebraic structure of word senses, with applications to polysemy.” *Transactions of the Association for Computational Linguistics*, 6, 2018, pp. 483–495.
- [41] V. Varma, R. Shah, Z. Kenton, J. Kramár, and R. Kumar. “Explaining grokking through circuit efficiency.” *arXiv preprint arXiv:2309.02390*, 2023.
- [42] Y. LeCun. *The MNIST database of handwritten digits*. 1998. URL: <http://yann.lecun.com/exdb/mnist/>.
- [43] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. “The llama 3 herd of models.” *arXiv preprint arXiv:2407.21783*, 2024.
- [44] S. Xu, L. Jiayao, Z. He, C. Peng, and W. Xu. “The Surprising Effectiveness of Randomness in LLM Pruning.” In: *Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference*.
- [45] Y. Li, E. J. Michaud, D. D. Baek, J. Engels, X. Sun, and M. Tegmark. “The geometry of concepts: Sparse autoencoder feature structure.” *Entropy*, 27(4), 2025, p. 344.
- [46] A. Templeton et al. “Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet.” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html>.

- [47] K. Meng, D. Bau, A. Andonian, and Y. Belinkov. “Locating and editing factual associations in gpt.” *Advances in neural information processing systems*, 35, 2022, pp. 17359–17372.
- [48] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. “Think you have solved question answering? try arc, the ai2 reasoning challenge.” *arXiv preprint arXiv:1803.05457*, 2018.
- [49] N. Li, A. Pan, A. Gopal, S. Yue, D. Berrios, A. Gatti, J. D. Li, A.-K. Dombrowski, S. Goel, L. Phan, et al. “The wmdp benchmark: Measuring and reducing malicious use with unlearning.” *arXiv preprint arXiv:2403.03218*, 2024.
- [50] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. “Fitnets: Hints for thin deep nets.” *arXiv preprint arXiv:1412.6550*, 2014.
- [51] S. Sun, Y. Cheng, Z. Gan, and J. Liu. “Patient knowledge distillation for bert model compression.” *arXiv preprint arXiv:1908.09355*, 2019.
- [52] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. “Tinybert: Distilling bert for natural language understanding.” *arXiv preprint arXiv:1909.10351*, 2019.
- [53] V. Sanh. “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.” *arXiv preprint arXiv:1910.01108*, 2019.
- [54] J. McCarley, R. Chakravarti, and A. Sil. “Structured pruning of a bert-based question answering model.” *arXiv preprint arXiv:1910.06360*, 2019.
- [55] A. Meta. “Llama 3.2: Revolutionizing edge AI and vision with open, customizable models.” *Meta AI Blog*. Retrieved March 2025, 2024.
- [56] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova. “Well-read students learn better: On the importance of pre-training compact models.” *arXiv preprint arXiv:1908.08962*, 2019.
- [57] Y. LeCun, J. S. Denker, and S. A. Solla. “Optimal Brain Damage.” In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan Kaufmann, 1990, pp. 598–605. URL: <https://proceedings.neurips.cc/paper/1989/hash/6c9882bbac1c7093bd25041881277658-Abstract.html>.
- [58] B. Hassibi and D. G. Stork. “Second Order Derivatives for Network Pruning: Optimal Brain Surgeon.” In: *Advances in Neural Information Processing Systems 5*. Ed. by S. J. Hanson, J. D. Cowan, and C. L. Giles. Morgan Kaufmann, 1993, pp. 164–171. URL: <https://proceedings.neurips.cc/paper/1992/hash/647-second-order-derivatives-for-network-pruning-optimal-brain-surgeon>.
- [59] V. Sanh, T. Wolf, and A. Rush. “Movement pruning: Adaptive sparsity by fine-tuning.” *Advances in neural information processing systems*, 33, 2020, pp. 20378–20389.
- [60] M. Xia, T. Gao, Z. Zeng, and D. Chen. “Sheared llama: Accelerating language model pre-training via structured pruning.” *arXiv preprint arXiv:2310.06694*, 2023.
- [61] X. Ma, G. Fang, and X. Wang. “Llm-pruner: On the structural pruning of large language models.” *Advances in neural information processing systems*, 36, 2023, pp. 21702–21720.
- [62] M. Xia, Z. Zhong, and D. Chen. “Structured pruning learns compact and accurate models.” *arXiv preprint arXiv:2204.00408*, 2022.

- [63] N. M. Vural and M. A. Erdogdu. *Pruning is Optimal for Learning Sparse Features in High-Dimensions*. 2024. arXiv: [2406.08658](https://arxiv.org/abs/2406.08658) [stat.ML]. URL: <https://arxiv.org/abs/2406.08658>.
- [64] Z. Liu, Y. Liu, E. J. Michaud, J. Gore, and M. Tegmark. “Physics of Skill Learning.” *arXiv preprint arXiv:2501.12391*, 2025.
- [65] M. Chen, N. Roberts, K. Bhatia, J. Wang, C. Zhang, F. Sala, and C. Ré. “Skill-it! a data-driven skills framework for understanding and training language models.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 36000–36040.
- [66] Y. Cao and J. Yang. “Towards making systems forget with machine unlearning.” In: *2015 IEEE symposium on security and privacy*. IEEE. 2015, pp. 463–480.
- [67] L. Bourtole, V. Chandrasekaran, C. A. Choquette-Choo, H. Jia, A. Travers, B. Zhang, D. Lie, and N. Papernot. “Machine Unlearning.” *arXiv preprint arXiv:1912.03817*, 2019. DOI: [10.48550/arXiv.1912.03817](https://arxiv.org/abs/1912.03817). URL: <https://arxiv.org/abs/1912.03817>.
- [68] Y. Yao, X. Xu, and Y. Liu. “Large language model unlearning.” *Advances in Neural Information Processing Systems*, 37, 2024, pp. 105425–105475.
- [69] J. Chen and D. Yang. “Unlearn what you want to forget: Efficient unlearning for llms.” *arXiv preprint arXiv:2310.20150*, 2023.
- [70] K. Gu, M. R. U. Rashid, N. Sultana, and S. Mehnaz. “Second-order information matters: Revisiting machine unlearning for large language models.” *arXiv preprint arXiv:2403.10557*, 2024.
- [71] P. Guo, A. Syed, A. Sheshadri, A. Ewart, and G. K. Dziugaite. *Mechanistic Unlearning: Robust Knowledge Unlearning and Editing via Mechanistic Localization*. 2024. arXiv: [2410.12949](https://arxiv.org/abs/2410.12949) [cs.LG]. URL: <https://arxiv.org/abs/2410.12949>.
- [72] B. W. Lee, A. Foote, A. Infanger, L. Shor, H. Kamath, J. Goldman-Wetzler, B. Woodworth, A. Cloud, and A. M. Turner. *Distillation Robustifies Unlearning*. 2025. arXiv: [2506.06278](https://arxiv.org/abs/2506.06278) [cs.LG]. URL: <https://arxiv.org/abs/2506.06278>.
- [73] A. Cloud, J. Goldman-Wetzler, E. Wybitul, J. Miller, and A. M. Turner. “Gradient routing: Masking gradients to localize computation in neural networks.” *arXiv preprint arXiv:2410.04332*, 2024.
- [74] Y. Li, P. Zhao, G. Yuan, X. Lin, Y. Wang, and X. Chen. “Pruning-as-search: Efficient neural architecture search via channel pruning and structural reparameterization.” *arXiv preprint arXiv:2206.01198*, 2022.
- [75] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. “Zoom In: An Introduction to Circuits.” *Distill*, 2020. <https://distill.pub/2020/circuits/zoom-in>. DOI: [10.23915/distill.00024.001](https://doi.org/10.23915/distill.00024.001).
- [76] S. Marks, C. Rager, E. J. Michaud, Y. Belinkov, D. Bau, and A. Mueller. “Sparse feature circuits: Discovering and editing interpretable causal graphs in language models.” *arXiv preprint arXiv:2403.19647*, 2024.
- [77] J. Lindsey et al. “On the Biology of a Large Language Model.” *Transformer Circuits Thread*, 2025. URL: <https://transformer-circuits.pub/2025/attribution-graphs/biology.html>.

- [78] E. J. Michaud, I. Liao, V. Lad, Z. Liu, A. Mudide, C. Loughridge, Z. C. Guo, T. R. Kheirkhah, M. Vukelić, and M. Tegmark. “Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code.” *Entropy*, 26(12), 2024, p. 1046.

3.8 Appendix

3.8.1 Additional results on CMSP

Here we include some additional results on CMSP. We first explore whether network shape influences learning dynamics when learning composite tasks (w/ corresponding atomic tasks in the training dataset) vs. when learning an atomic task on its own. We find that the effect of depth is similar between composite vs. atomic tasks. This suggests that our networks do not necessarily need to be deep in order to take advantage of compositional structure in this task. Note that a previous version of this manuscript suggested the opposite—we had observed that deeper networks learn composite CMSP tasks faster than shallow networks, but it turns out that deeper networks also learn standalone atomic tasks faster than shallow networks. We show loss curves of depth-3 width-128, depth-2 width-361, and depth-2 width-128 networks in Figure 3.7.

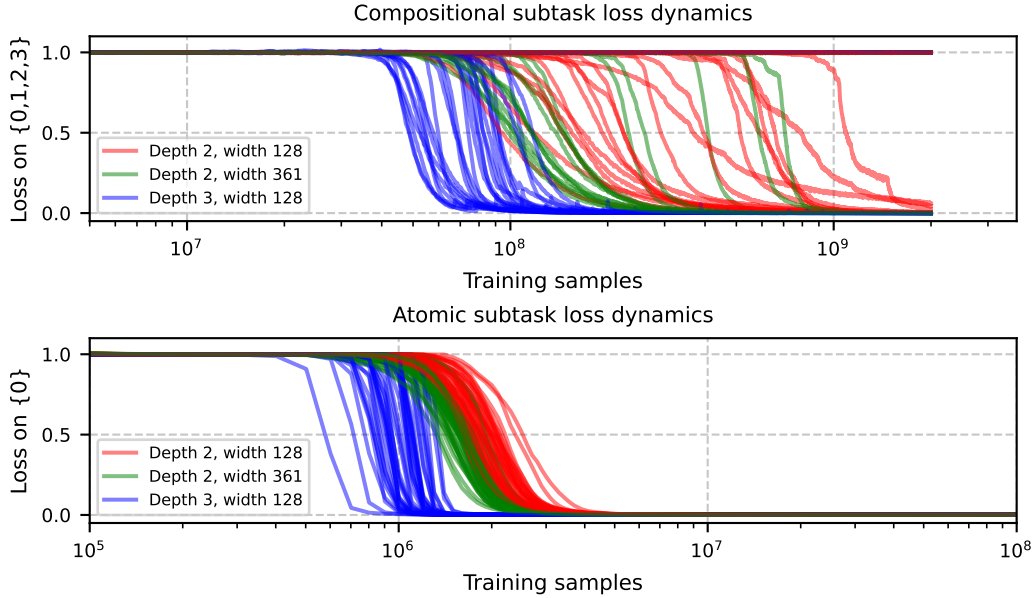


Figure 3.7: Impact of network shape on CMSP learning. **Top:** Forty runs of training with different seeds on CMSP dataset $\mathcal{D}_{\{0\}} \cup \mathcal{D}_{\{1\}} \cup \mathcal{D}_{\{2\}} \cup \mathcal{D}_{\{3\}} \cup \mathcal{D}_{\{0,1,2,3\}}$ with $m = 4$, $n = 64$, $k = 4$. We find that 27/40 runs converge with two hidden layers versus 19/40 runs with a single hidden layer with width 128 and 16/40 runs with a single hidden layer and width 361. Single-hidden-layer networks are therefore able to learn composite subtasks in CMSP roughly as well as deeper networks. **Bottom:** Forty runs of training on a standalone atomic task $m = 1$, $n = 64$, $k = 4$.

We also show how ablation scores vary across neurons in our unregularized networks. With the setup in Section 3.3.3, we show ablation scores for each neuron for each subtask in Figure 3.8

and Figure 3.9. We find that there are neurons which have high scores across most subtasks, even subtasks in different “skill trees” ($\{0,1,2\}$ versus $\{3,4,5\}$).

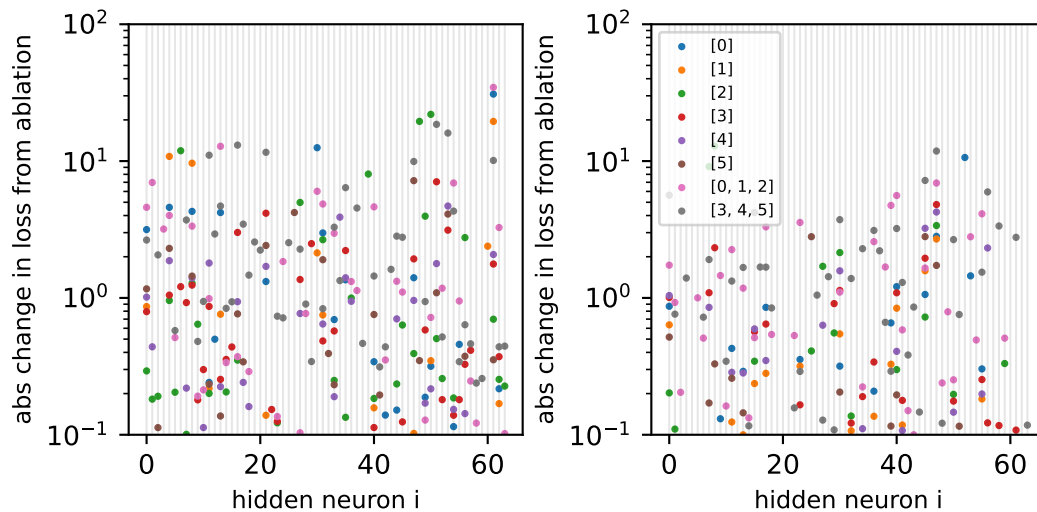


Figure 3.8: For each hidden neuron in the first hidden layer (**left**) and the second hidden layer (**right**) in a CMSP network, we show the ablation score of that neuron for each subtask. For some neurons, ablation scores are high across multiple subtasks, even from separate “skill trees”.

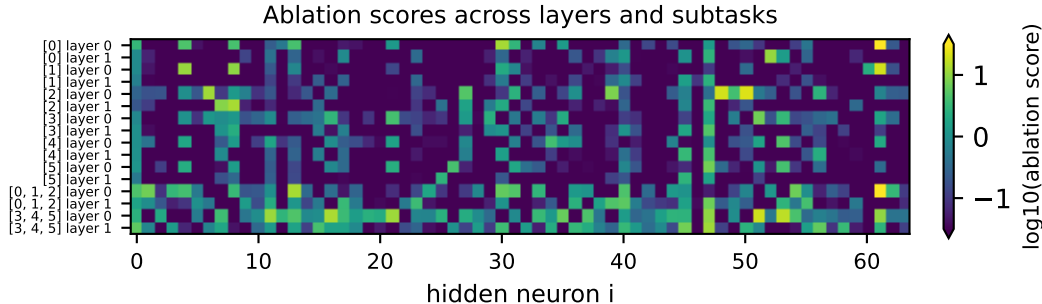


Figure 3.9: Ablation scores for each subtask and each neuron. This is simply a different way of visualizing the results in Figure 3.8.

In Figure 3.10 we show sparsity vs accuracy curves from pruning like in Figure 3.3 (bottom left), across seeds and network sizes. In our unregularized networks, there is a lot of variation between runs in how entangled the different subtasks are.

3.8.2 Additional results on LLMs

Additional plots

We also show some additional results with LLMs. In Figure 3.11, we show how attribution scores correlate with ablation scores in LLMs. We show results for pruning both neurons and residual stream dimensions.

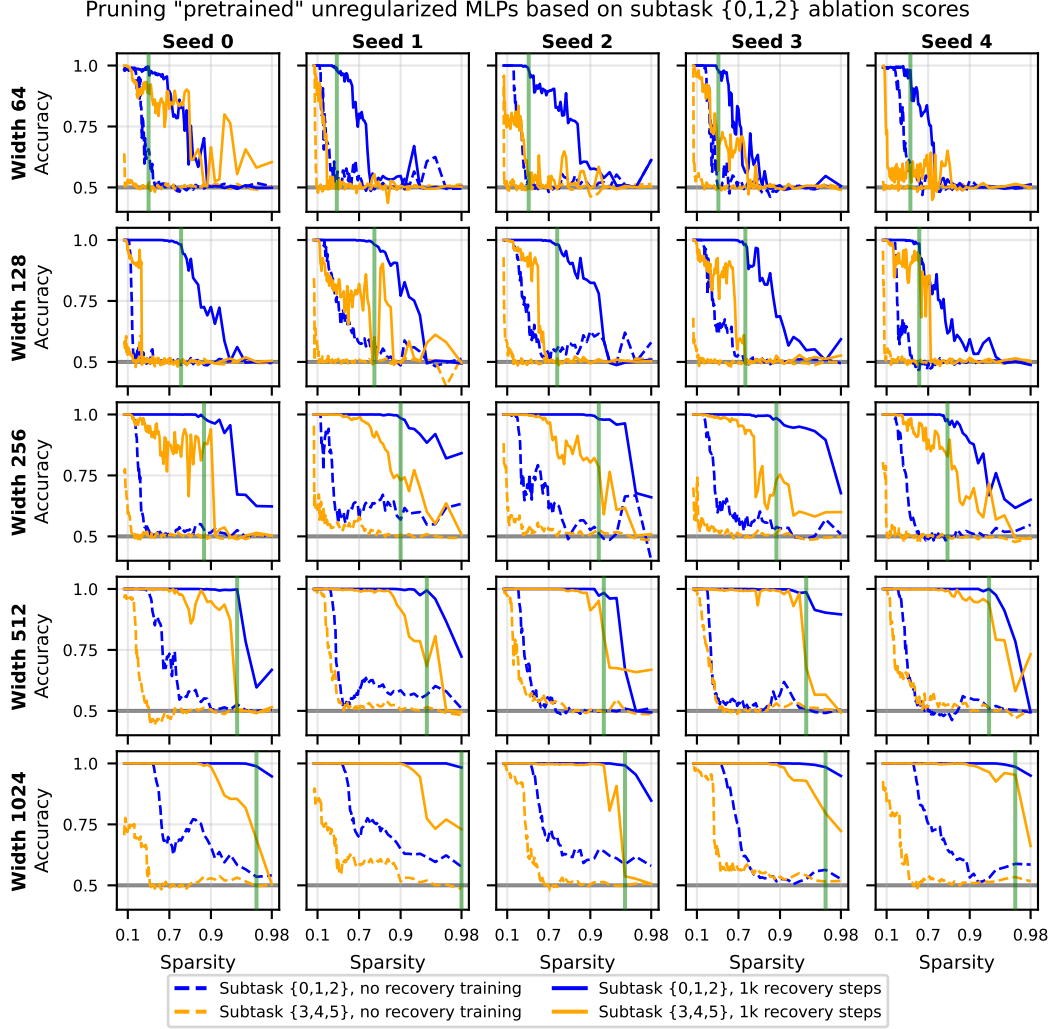


Figure 3.10: Pruning sparsity vs accuracy curves for pretrained CMSP networks. We train MLPs of varying width and random seed on a CMSP dataset with two skill trees, as in Figure 3.3. We prune neurons based on ablation score on subtask {0,1,2}. On some networks, ablating neurons to maximum sparsity while preserving performance on subtask {0,1,2} robustly unlearns subtask {3,4,5}. However, for other networks, the subtasks seem more entangled, and we can recover performance on subtask {3,4,5} even after ablating to the maximum extent we can still recover 98% accuracy on subtask {0,1,2} (green line).

In Figure 3.12, we show learning curves across runs when training from scratch on Python documents, when performing distillation, and when performing recovery training after pruning both neurons and residual stream dimensions, to varying target sparsities, of Llama-3.2-1B.

In Figure 3.13, we show recovery curves after pruning neurons and residual stream dimensions from Llama-3.2-1B to varying sparsity levels. We find that pruning random components performs roughly as well as pruning components with the lowest attribution scores.

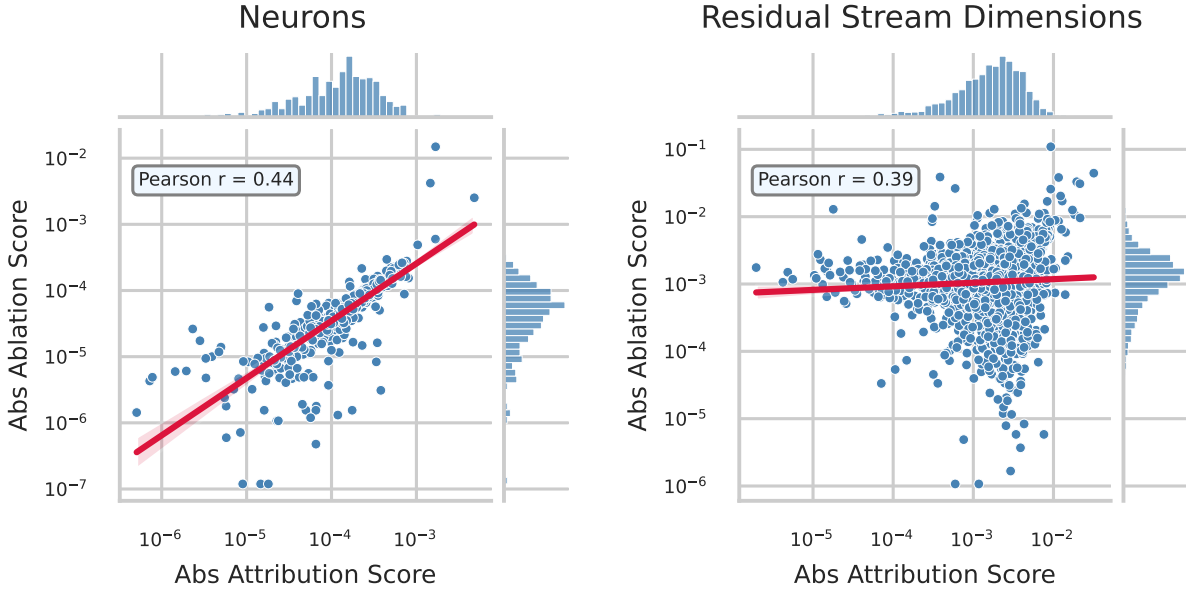


Figure 3.11: Comparison of ablation vs. attribution scores for Llama-3.2-1B neurons (left) and residual stream dimensions (right), evaluated on a single batch of Python code documents. For each figure, we fully ablate model components and compare the absolute change in loss (Abs Ablation Score, simply called “ablation score” elsewhere in the paper) with the absolute value of the linear estimate of this change computed from model gradients (Abs Attribution Score, called “attribution score” elsewhere). Note that the correlation between the log of these scores is 0.80 for neurons and 0.05 for residual stream components.

Additional experimental details

When training networks from scratch and distilling from scratch (Figure 3.12 and Figure 3.6, we train transformers with the Llama 3 architecture [43] of varying size, listed in Table 3.2.

Table 3.2: Transformer model configurations explored

Hidden size	#Layers	#Heads	Intermediate size
256	4	4	1,024
512	8	8	2,048
768	12	12	3,072
864	16	16	3,456
1,024	16	16	4,096
1,280	20	20	5,120
1,536	24	24	6,144
1,792	28	28	7,168
2,048	32	32	8,192

When we prune Llama-3.2-1B, we prune neurons and residual stream dimensions with the sparsity combinations shown in Table 3.3.

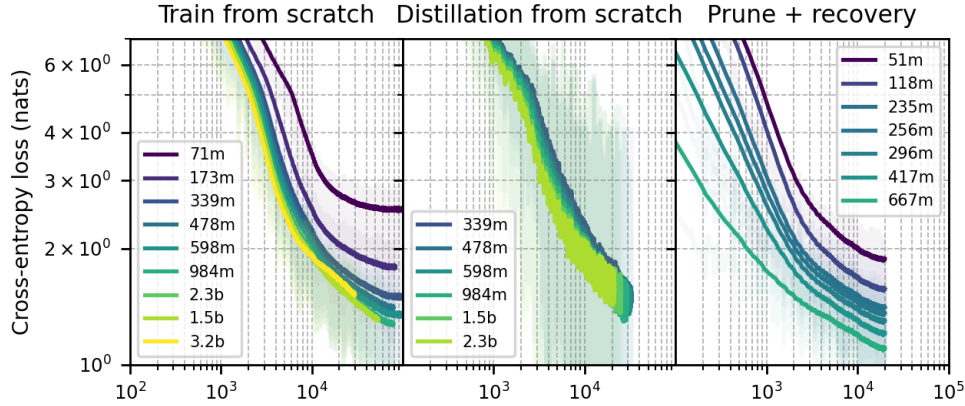


Figure 3.12: Training curves of LLMs on Python documents when training from scratch (**left**), training from scratch but with distillation of a larger pretrained LLM (**center**), and when training pruned pretrained models (**right**) to recover performance lost during training.

Table 3.3: Neuron and residual sparsity configurations when using attribution pruning on Llama-3.2-1B.

Neuron sparsity	Residual sparsity
0.50	0.50
0.80	0.50
0.90	0.50
0.95	0.50
0.80	0.80
0.90	0.90

3.8.3 Compute estimates

We estimate the compute used for each of our experiments.

For the CMSP training experiments shown in Figure 3.2 and Figure 3.7, we ran 4 configurations each with 40 different seeds. We ran each job on a GPU, but on a cluster with a variety of different node configurations and GPUs. Jobs generally took between 5-60 minutes to complete, for between 13-160 total hours of GPU time.

For the experiments shown in Figure 3.3 and Figure 3.10, each run, across 5 choices of seed and 5 choices of width, trained networks on a CMSP task, ran pruning experiments, performed group-lasso training, and pruned again. Each such run took between 5-120 minutes on our cluster with varying GPU configurations, for a total time of 2-50 hours of GPU time.

For our MNIST experiments shown in Figure 3.4, we plot a total of 54 points, each of which is averaged over 10 training runs. We estimate that each run took around 2 minutes on our cluster, totaling to around 18 hours of GPU time.

For our LLM experiments, we ran our experiments on A100-80GB nodes, with a single GPU allocated per experiment. When we trained models from scratch on Python documents, we used 9 configurations with job lengths between 1-3 days. For distillation we had 9 configurations with

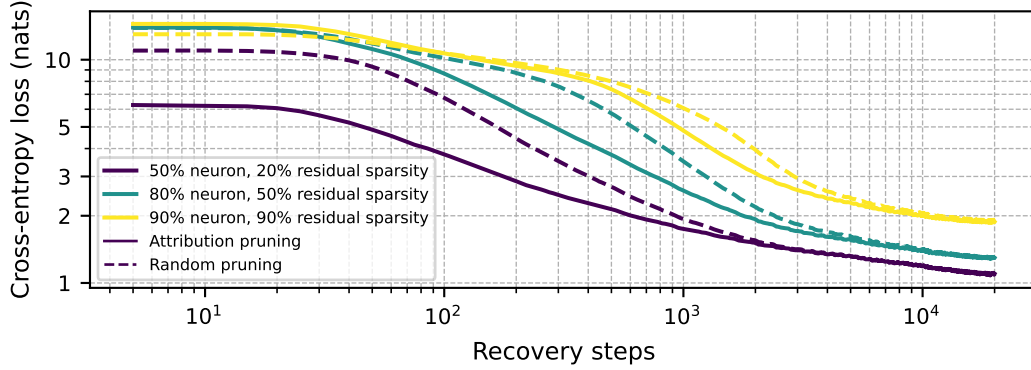


Figure 3.13: Recovery training curves after pruning neurons and residual stream dimensions, for Llama-3.2-1B on Python code documents. We compare recovery performance when pruning based on attribution scores vs. choosing components randomly. Attribution scores are computed across 1024 documents with a max length of 1024 tokens. We find that pruning with attribution scores is better than pruning random components, however this gap is eventually recovered. For instance, at step 5365 when our run with 50% neuron sparsity and 20% residual sparsity first matches the performance of the original model (≈ 1.3 nats), the performance on randomly-pruned model is almost identical at 1.301 nats.

job lengths between 2-3 days, though some jobs failed. For the group lasso training experiments in Figure 3.5, we show 3 configurations trained with job lengths of 3 days, and we performed recovery training on these models with a job length of 1 day. We estimate that the total time for these jobs was less than 1600 hours of A100 time, though the full set of experiments we attempted in the work that led to this manuscript could be over 5000 hours.

Chapter 4

Precision Machine Learning^{*}

So far, we have considered models of learning and scaling where the “parts” that networks learn are discrete, and scaling laws come from the statistics of these parts. However, there is a different way of understanding neural network scaling laws via an argument from approximation theory. The idea is that neural networks can be viewed as simply fitting a function defined on a manifold $\mathbb{R}^n \supseteq M \rightarrow \mathbb{R}$. As networks are scaled up, they can more precisely fit this function, and the scaling exponent is determined by the dimension of the manifold. In this paper, we see that modularity can play an important part of stories like this too. The basic idea is that if the function we want to learn *composes* lower-dimensional functions (modules), then networks can optimally scale as if the dimension of the data manifold was the maximum input dimension of each module.

Abstract

We explore unique considerations involved in fitting machine learning (ML) models to data with very high precision, as is often required for science applications. We empirically compare various function approximation methods and study how they *scale* with increasing parameters and data. We find that neural networks (NNs) can often outperform classical approximation methods on high-dimensional examples, by auto-discovering and exploiting modular structures therein. However, neural networks trained with common optimizers are less powerful for low-dimensional cases, which motivates us to study the unique properties of neural network loss landscapes and the corresponding optimization challenges that arise in the high precision regime. To address the optimization issue in low dimensions, we develop training tricks which enable us to train neural networks to extremely low loss, close to the limits allowed by numerical precision.

^{*}Eric J. Michaud, Ziming Liu, and Max Tegmark. “Precision machine learning.” *Entropy* 25, no. 1 (2023): 175.

4.1 Introduction

Most machine learning practitioners do not need to fit their data with much precision. When applying machine learning to traditional tasks in artificial intelligence such as those in computer vision or natural language processing, one typically does not desire to bring training loss all the way down to exactly zero, in part because training loss is just a proxy for some other performance measure like accuracy that one actually cares about, or because there is intrinsic uncertainty which makes perfect prediction impossible, e.g., for language modeling. Accordingly, to save memory and speed up computation, much work has gone into *reducing* the numerical precision used in models without sacrificing model performance much [1–3]. However, modern machine learning methods, and deep neural networks in particular, are now increasingly being applied to science problems, for which being able to fit models *very* precisely to (high-quality) data can be important [4–7]. Small absolute changes in loss can make a big difference, e.g., for the symbolic regression task of identifying an exact formula from data [8,9].

It is therefore timely to consider what, if any, unique considerations arise when attempting to fit ML models very precisely to data, a regime we call *Precision Machine Learning (PML)*. How does pursuit of precision affect choice of method? How does optimization change in the high-precision regime? Do otherwise-obscure properties of model expressivity or optimization come into focus when one cares a great deal about precision? In this paper, we explore these basic questions.

4.1.1 Problem Setting

We study regression in the setting of supervised learning, in particular the task of fitting functions $f : \mathbb{R}^d \rightarrow \mathbb{R}$ to a dataset of $D = \{(\vec{x}_i, y_i = f(\vec{x}_i))\}_{i=1}^{|D|}$. In this work, we mostly restrict our focus to functions f which are given by symbolic formulas. Such functions are appropriate for our purpose, of studying precision machine learning for science applications, since they (1) are ubiquitous in science, fundamental to many fields’ descriptions of nature, (2) are precise, not introducing any intrinsic noise in the data, making extreme precision possible, and (3) often have interesting structure such as *modularity* that sufficiently clever ML methods should be able to discover and exploit. We use a dataset of symbolic formulas from [10], collected from the Feynman Lectures on Physics [11].

Just how closely can we expect to fit models to data? When comparing a model prediction $f_\theta(\vec{x}_i)$ to a data point y_i , the smallest nonzero difference allowed is determined by the numerical precision used. IEEE 754 64-bit floats [12] have 52 mantissa bits, so if y_i and $f_\theta(\vec{x}_i)$ are of order unity, then the smallest nonzero difference between them is $\epsilon_0 = 2^{-52} \sim 10^{-16}$. We should not expect to achieve *relative RMSE loss* below 10^{-16} , where relative RMSE loss, on a dataset D , is:

$$\ell_{\text{rms}} \equiv \left(\frac{\sum_{i=1}^{|D|} |f_\theta(\vec{x}_i) - y_i|^2}{\sum_{i=1}^{|D|} y_i^2} \right)^{\frac{1}{2}} = \frac{|f_\theta(\vec{x}_i) - y_i|_{\text{rms}}}{y_{\text{rms}}}. \quad (4.1)$$

In practice, precision can be bottlenecked earlier by the computations performed within the model f_θ . The task of precision machine learning is to try to push the loss down many orders of magnitude, driving ℓ_{rms} as close as possible to the numerical noise floor ϵ_0 .

4.1.2 Decomposition of Loss

One can similarly define *relative MSE loss* $\ell_{\text{mse}} \equiv \ell_{\text{rms}}^2$, as well as non-relative (standard) MSE loss $L_{\text{mse}}(f) = \frac{1}{|D|} \sum_{i=1}^D (f_{\theta}(\vec{x}_i) - y_i)^2$, and $L_{\text{rms}} = \sqrt{L_{\text{mse}}}$. Minimizing ℓ_{rms} , ℓ_{mse} , L_{rms} , L_{mse} are equivalent up to numerical errors. Note that (relative) expected loss can be defined on a probability distribution $\mathbb{P}_{(\mathbb{R}^d, \mathbb{R})}$, like so:

$$\ell^{\mathbb{P}}_{\text{rms}} = \left(\frac{\mathbb{E}_{(\vec{x}, y) \sim \mathbb{P}}[(f_{\theta}(\vec{x}) - y)^2]}{\mathbb{E}_{(\vec{x}, y) \sim \mathbb{P}}[y^2]} \right)^{\frac{1}{2}}. \quad (4.2)$$

When we wish to emphasize the distinction between loss on a dataset D (empirical loss) and a distribution $\mathbb{P}$ (expected loss), we write ℓ^D and $\ell^{\mathbb{P}}$. In the spirit of [13], we find it useful to decompose sources of error into different sources, which we term *optimization error*, *sampling luck*, the *generalization gap*, and *architecture error*. A given model architecture parametrizes a set of expressible functions $\mathcal{H}$. One can define three functions of interest within $\mathcal{H}$:

$$f_{\mathbb{P}}^{\text{best}} \equiv \underset{f \in \mathcal{H}}{\operatorname{argmin}} \{ \ell^{\mathbb{P}}(f) \}, \quad (4.3)$$

the best model on the expected loss $\ell^{\mathbb{P}}$,

$$f_D^{\text{best}} \equiv \underset{f \in \mathcal{H}}{\operatorname{argmin}} \{ \ell^D(f) \}, \quad (4.4)$$

the best model on the empirical loss ℓ^D , and

$$f_D^{\text{used}} = \mathcal{A}(\mathcal{H}, D, L), \quad (4.5)$$

the model found by a given learning algorithm $\mathcal{A}$ which performs possibly imperfect optimization to minimize empirical loss L on D .

We can therefore decompose the empirical loss as follows:

$$\begin{aligned} \ell^D(f_D^{\text{used}}) = & \underbrace{[\ell^D(f_D^{\text{used}}) - \ell^D(f_D^{\text{best}})]}_{\text{optimization error}} + \underbrace{[\ell^D(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f_D^{\text{best}})]}_{\text{sampling luck}} + \\ & \underbrace{[\ell^{\mathbb{P}}(f_D^{\text{best}}) - \ell^{\mathbb{P}}(f^{\text{best}}_{\mathbb{P}})]}_{\text{generalization gap}} + \underbrace{\ell^{\mathbb{P}}(f^{\text{best}}_{\mathbb{P}})}_{\text{architecture error}}, \end{aligned} \quad (4.6)$$

where all terms are positive except possibly the *sampling luck*, which is zero on average, has a standard deviation shrinking with data size $|D|$ according to the Poisson scaling $|D|^{-1/2}$, and will be ignored in the present paper. The generalization gap has been extensively studied in prior work, so this paper will focus exclusively on the optimization error and the architecture error.

To summarize: the architecture error is the best possible performance that a given architecture can achieve on the task, the generalization gap is the difference between the optimal performance on the training set D and the architecture error, and the optimization error is the error introduced by imperfect optimization – the difference between the error on the training set found by imperfect optimization and the optimal error on the training set. When comparing methods and studying their scaling, it useful to ask which of these error sources dominate. We will see that both architecture error and optimization error can be quite important in the high-precision regime, as we will elaborate on in Sections 4.2-4.3 and Section 4.4, respectively.

4.1.3 Importance of Scaling Exponents

In this work, one property that we focus on is how methods *scale* as we increase parameters or training data. This builds on a recent body of work on scaling laws in deep learning [14–22] which has found that, on many tasks, loss decreases predictably as a power-law in the number of model parameters and amount of training data. Attempting to understand this scaling behavior, [23,24] argue that in some regimes, cross-entropy and MSE loss should scale as $N^{-\alpha}$, where $\alpha \gtrsim 4/d$, N is the number of model parameters, and d is the *intrinsic dimensionality* of the *data manifold* of the task.

Consider the problem of approximating some analytic function $f : [0, 1]^d \rightarrow \mathbb{R}$ with some function which is a piecewise n -degree polynomial. If one partitions a hypercube in $\mathbb{R}^d$ into regions of length ϵ and approximates f as a n -degree polynomial in each region (requiring $N = O(1/\epsilon^d)$ parameters), absolute error in each region will be $O(\epsilon^{n+1})$ (given by the degree- $(n+1)$ term in the Taylor expansion of f) and so absolute error scales as $N^{-\frac{n+1}{d}}$. If neural networks use ReLU activations, they are piecewise linear ($n = 1$) and so we may expect $\text{rmse}(N) \propto N^{-\frac{2}{d}}$. Sharma & Kaplan [23] argue that if neural networks map input data onto an intermediate representation, where representations lie on a manifold of intrinsic dimension $d^* < d$, and then perform a piecewise linear fit to a function on this manifold, then RMSE error should scale as N^{-2/d^*} .

If one desires very low loss, then the exponent α , the rate at which methods approach their best possible performance¹ matters a great deal. Sharma & Kaplan [23] note that $4/d$ ($2/d$ for RMSE loss) is merely a lower-bound on the scaling rate – we consider ways that neural networks can improve on this bound. Understanding model scaling is key to understanding the feasibility of achieving high precision.

4.1.4 Organization

This paper is organized as follows: In Section 4.2 we discuss piecewise linear approximation methods, comparing ReLU networks with linear simplex interpolation. We discover that neural networks can sometimes substantially outperform linear simplex interpolation, and hypothesize that NNs do this by exploiting the modularity of the problem, which we call the *modularity hypothesis*. In Section 4.3 we discuss nonlinear methods, including neural networks with nonlinear activation functions, and find that optimization error, rather than approximation error, can be a major bottleneck to achieving high-precision fits. In Section 4.4 we discuss the optimization challenge of high-precision neural network training – how optimization difficulties can often make total error far worse than the limits of what architecture error allows. We attempt to develop optimization methods for overcoming these problems and describe their limitations, then conclude in Section 4.5.

¹The best possible performance can be determined either by precision limits or by noise intrinsic to the problem, such as intrinsic entropy of natural language.

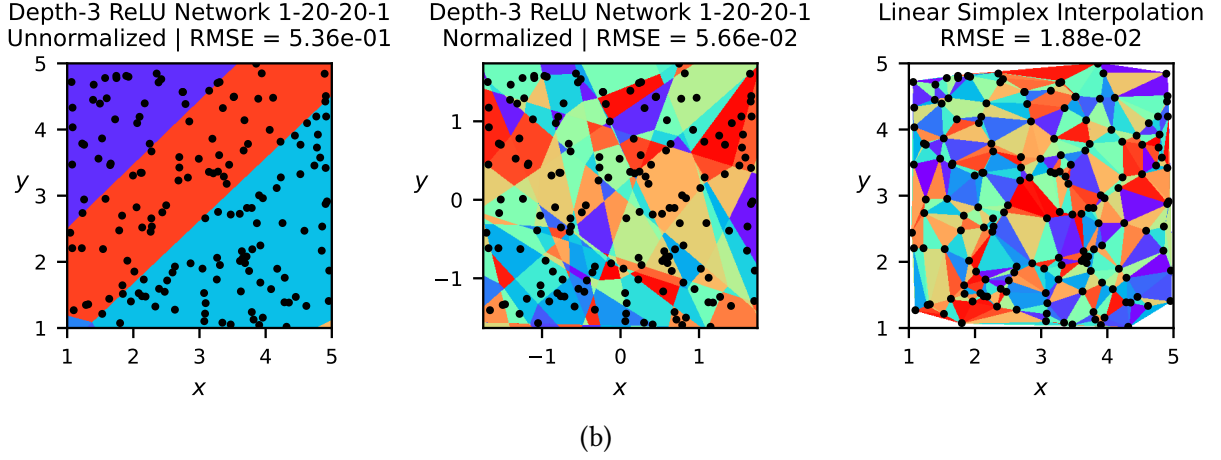
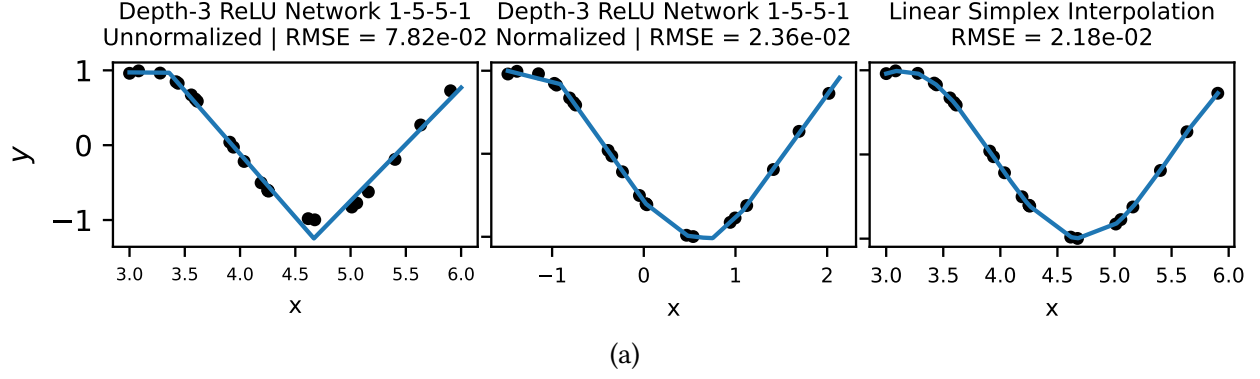


Figure 4.1: In (a) (top), we show the solutions learned by a ReLU network and linear simplex interpolation on the 1D problem $y = \cos(2x)$. In (b) (bottom), we visualize linear regions for a ReLU network, trained on unnormalized data (left) and normalized data (center), as well as linear simplex interpolation (right) on the 2D problem $z = xy$. In general, we find that normalizing data to have zero mean and unit variance improves network performance, but that linear simplex interpolation outperforms neural networks on low-dimensional problems by better vertex placement.

4.2 Piecewise Linear Methods

We first consider approximation methods which provide a piecewise linear fit to data. We focus on two such methods: linear simplex interpolation and neural networks with ReLU activations.

To review, linear simplex interpolation works as follows: given our dataset of $|D|$ input-output pairs $\{(\vec{x}_i, y_i)\}_{i=1}^{|D|}$, linear simplex interpolation first computes a Delaunay triangulation from $\vec{x}_1, \dots, \vec{x}_{|D|}$ in the input space $\mathbb{R}^d$, partitioning the space into a collection of d -simplices, each with $d + 1$ vertices, whose union is the convex hull of the input points. Since $d + 1$ points determine a linear (affine) function $\mathbb{R}^d \rightarrow \mathbb{R}$, the function f can be approximated within each d -simplex as the unique linear function given by the value of the function f at the vertices. This gives a piecewise linear function on the convex hull of the training points. Linear simplex interpolation needs to store $N = |D|(d + 1)$ parameters: $|D|d$ values for the vertices $\vec{x}_i$, and $|D|$ values for the corresponding function values y_i .

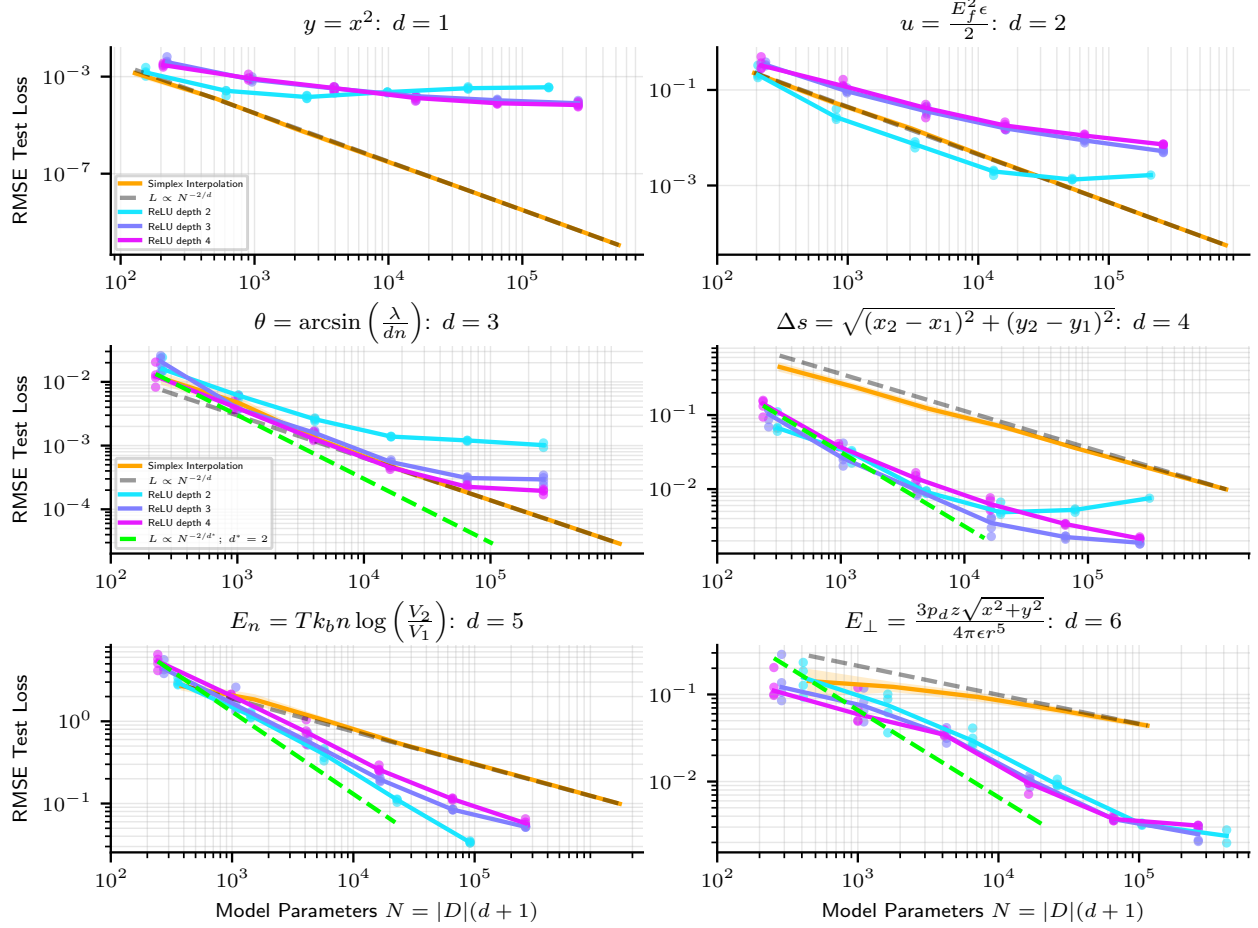


Figure 4.2: Scaling of linear simplex interpolation versus ReLU NNs. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, we find that NNs sometimes scale better (at least in early regimes) as N^{-2/d^*} , where $d^* = 2$, on high dimensional problems.

Neural networks with ReLU activations also give a piecewise linear fit f_θ . We consider only fully-connected feedforward networks, a.k.a. multilayer perceptrons (MLPs). Such networks consist of a sequence of alternating affine transformations $T : \vec{x} \mapsto W\vec{x} + b$ and element-wise nonlinearities $\sigma(\vec{x})_i = \sigma(\vec{x}_i)$ for an activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$:

$$f_\theta = T_{k+1} \circ \sigma \circ T_k \circ \dots \circ T_2 \circ \sigma \circ T_1.$$

Following [25], we define the depth of the network as the number of affine transformations in the network, which is one greater than the number of hidden layers k . As shown in [25], any piecewise linear function on $\mathbb{R}^d$ can be represented by a sufficiently wide ReLU NN with at most $\lceil \log_2(d+1) \rceil + 1$ depth. Therefore, sufficiently wide and deep networks are able to exactly express functions given by linear simplex interpolation. A natural question then is: given the same amount of data and parameters, how do the two methods compare? We discover that simplex interpolation performs better on 1D and 2D problems, but that neural networks can outperform simplex interpolation on higher-dimensional problems. So although simplex interpolation and ReLU NNs both parametrize the same function class (piecewise linear functions), their performance

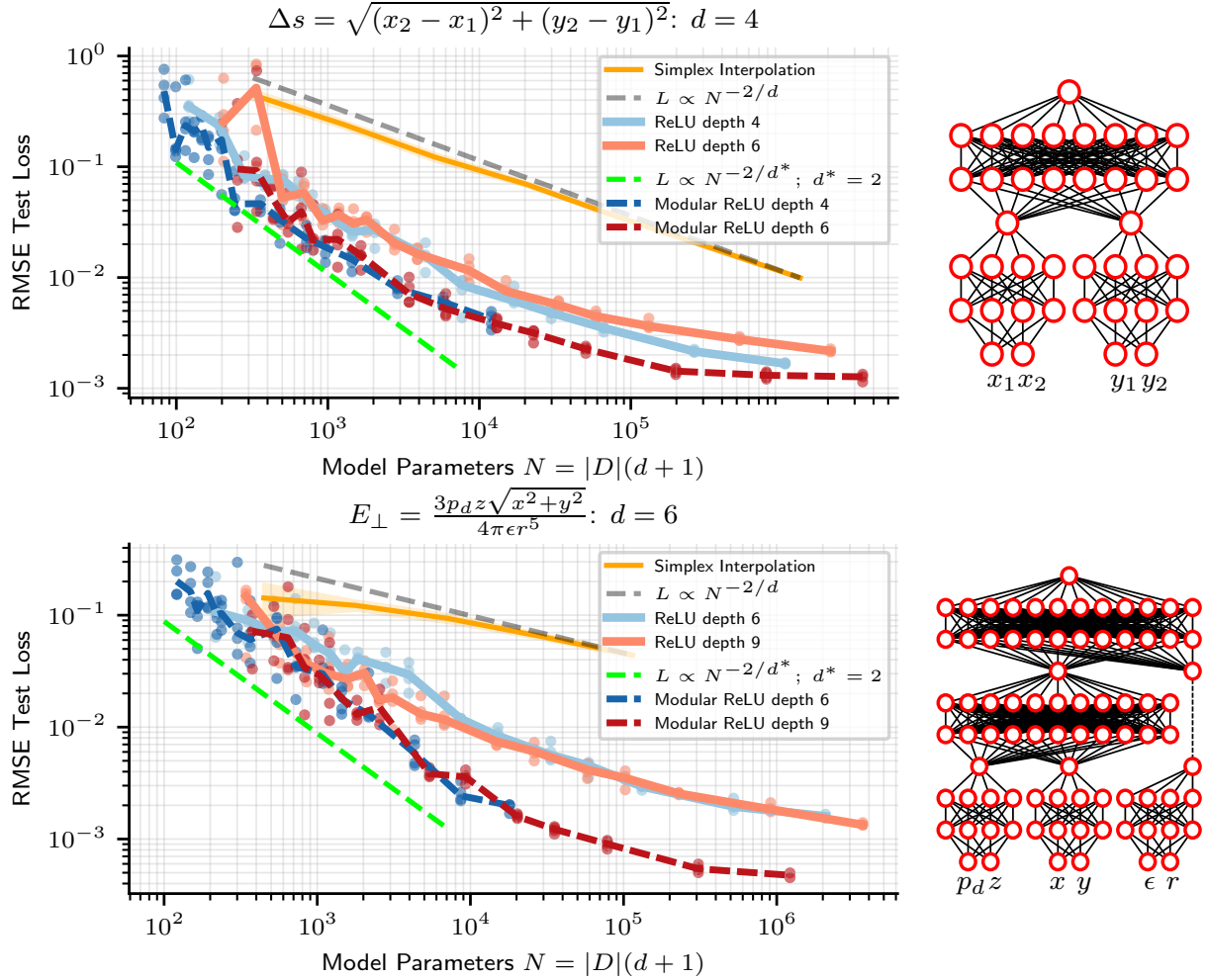


Figure 4.3: ReLU neural networks are seen to initially scale roughly as if they were modular. Networks with enforced modularity (dark blue and red, dashed line), with architecture depicted on the right, perform and scale similarly, though slightly better, than standard dense MLPs of the same depth (light blue and red).

can differ significantly in practice.²

In our experiments, we use the implementation of simplex interpolation from SciPy [27]. When training neural networks, we use the Adam optimizer [28] with a learning rate of 10^{-3} , and train for 20k steps. We use a batch size of $\min(|D|, 10^4)$. While we report loss using RMSE, we train using MSE loss. Training points are sampled uniformly from intervals specified by the AI Feynman dataset [10] (typically $[1, 5] \subset \mathbb{R}$ for each input), but when training neural networks, we normalize the input points [29] so that they have zero mean and unit variance along each dimension. We estimate test loss on datasets of 30k samples.³

In Section 4.1, we show for 1D and 2D problems the linear regions given both by simplex

²Ref [26] analyzed the expressivity of ReLU NNs in comparison with multivariate adaptive regression splines. Our work conducts a more extensive set of experiments to study how approximators *scale* in practice. We also restrict our focus to fitting physics functions, which have modular structure.

³Project code can be found at <https://github.com/ejmichaud/precision-ml>.

interpolation and by neural networks trained with a comparable number of parameters. For 2D problems, Figure 4.1 illustrates the importance of normalizing input data for ReLU networks. We see that there is a far higher density of linear regions around the data when input data is normalized, which leads to better performance. Neural networks, with the same number of parameters and trained with the same low-dimensional data, often have fewer linear regions than simplex interpolation.

In Figure 4.2, we show how the precision of linear simplex interpolation and neural networks scale empirically. Since simplex interpolation is a piecewise linear method, from the discussion in Section 4.1.3, we expect its RMSE error to scale as $N^{-2/d}$, and find that this indeed holds⁴. To provide a fair comparison with simplex interpolation when evaluating neural networks on a dataset of size D , we give it the same number of parameters $N = |D|(d + 1)$. From Figure 4.2, we see that simplex interpolation outperforms neural networks on low dimensional problems but that neural networks do better on higher-dimensional problems.

For the 1D example in Figure 4.2 (top left), we know that the amount by which the neural networks under-perform simplex interpolation is entirely due to optimization error. This is because any 1D piecewise linear function $f(x)$ with m corners at $x_1, \dots, x_m$ can trivially be written as a linear combination of m functions $\text{ReLU}(x - x_i)$.

We hypothesize that the reason why neural networks are able to beat simplex interpolation when fitting functions given by symbolic equations is that neural networks learn to exploit the inherent modularity of these problems. We term this the *modularity hypothesis*. This property of the regression functions we study, which we call *modularity*, has been referred to by other authors as “compositional sparsity” [30,31] or the property of being “hierarchically local compositional” [32]. Such functions consist of many low-dimensional functions composed with each other. For instance, given three two-dimensional functions f_1, f_2, f_3 , one can construct a modular four-dimensional function $g(x_1, x_2, x_3, x_4) = f_3(f_1(x_1, x_2), f_2(x_3, x_4))$. Many natural functions have this structure, notably those we are interested in given by symbolic equations, which compose together binary functions (+, −, ×, ÷) and unary functions (e^x , $\ln(x)$, etc.). We are particularly interested in modular functions where the computation graph describing the function has low *maximum arity* – that the constituent functions composing it individually have low input dimension. For symbolic equations, the maximum arity is two.

On modular problems, it is a theoretical possibility [33,34] that sufficiently deep neural networks could scale as if the problem dimension was the maximum arity of the problem computation graph, rather than the input dimension. This is possible if networks learn a modular solution to the problem, approximating each node in the computation graph and internally composing together these elements according to the computation graph. The error of ReLU NNs should therefore be able to scale as N^{-2/d^*} where d^* is the maximum-arity of the computation graph of the target function ($d^* = 2$ for symbolic equations). Scaling in this manner requires excess capacity to be allocated towards improving the approximation of all nodes in the graph, while continuing to compose them properly, rather than approximating a single function on a fixed data manifold.

We observe two tentative pieces of evidence for the modularity hypothesis as the reason why NNs beat linear simplex interpolation. First, we see from Figure 4.2, at least early in the scaling

⁴Scaling as $N^{-2/d}$ only holds when the model is evaluated on points not too close to the boundary of the training set. At the boundary, simplices are sometimes quite large, leading to a poor approximation of the target function close to the boundary, large errors, and worse scaling. In our experiments, we therefore compute test error only for points at least 10% (of the width of the training set in each dimension) from the boundary of the training set.

curves, that neural networks usually appear to scale not as $N^{-2/d}$, but rather as N^{-2/d^*} where $d^* = 2$, the maximum arity of the symbolic regression problems we train on. As an additional test, we train networks where we hard-code the modularity of the problem into the architecture, as depicted in Figure 4.3. Figure 4.3 indeed reveals how models for which we enforce the modularity of the problem perform and scale similarly to same-depth dense neural networks without modularity enforced. A modular architecture can be created from a dense one by forcing weight matrices to be block-diagonal (where we do not count off-diagonal entries towards the number of model parameters), but in practice we create modular architectures by creating a separate MLP for each node in the symbolic expression computation graph and connecting them together in accordance with the computation graph. See the diagrams in Figure 4.3 for an illustration of the modular architecture. In Figure 4.3, we plot modular and dense network performance against number of model parameters, but we also find that holding width constant, rather than number of parameters, modular networks still slightly outperform their dense counterparts. For instance, depth-6 width-100 modular networks outperform dense networks of the same width and depth, despite dense networks having $\approx 2.5\times$ fewer parameters. Such “less is more” results are to be expected if the optimal architecture is in fact modular, in which case a fully connected architecture wastes resources training large numbers of parameters that should be zero.

Our observation that neural networks sometimes appear to scale as if the problem dimension was the maximum arity of the computational graph generalizes and improves upon neural scaling results from Sharma & Kaplan [23], which argued that the effective problem dimension d^* is the intrinsic dimension of a fixed data manifold. In the view from [23], excess capacity is allocated towards approximating a function on this fixed manifold with greater resolution, rather than allocating capacity towards many internal modules and *composing* them in an appropriate manner as we have discussed. It is widely believed in the ML community that the dimensionality of the data manifold or the power-law statistics of the data covariance matrix spectra sets the neural scaling exponent (see for example the excellent treatment by Sharma & Kaplan [23] and Bahri et al. [24]), however, our results show that scaling exponents can be better under the modularity hypothesis, where networks exploit modularity to scale in the maximum arity of a problem’s computation graph.

4.3 Nonlinear Methods

We now turn our attention to approximation methods that are thoroughly nonlinear (as opposed to piecewise linear). As discussed in the introduction, methods approximating the target function f by a piecewise polynomial have a scaling exponent $\alpha = \frac{n+1}{d}$ where n is the degree of the polynomial.

In Figure 4.4, we plot the performance of approximation methods which are piecewise polynomial, for 1D, 2D and 3D problems. For 1D and 2D problems, we use splines of varying order. For 3D problems, we use the cubic spline interpolation method of [35]. We see empirically that these methods have scaling exponent $\alpha = (n + 1)/d$. If the order of the spline interpolator is high enough, and the dimension low enough, we see that relative RMSE loss levels out at $\epsilon_0 \approx 10^{-16}$ at the precision limit. Unfortunately, a basic limitation of these methods is that they are limited to low-dimensional problems.

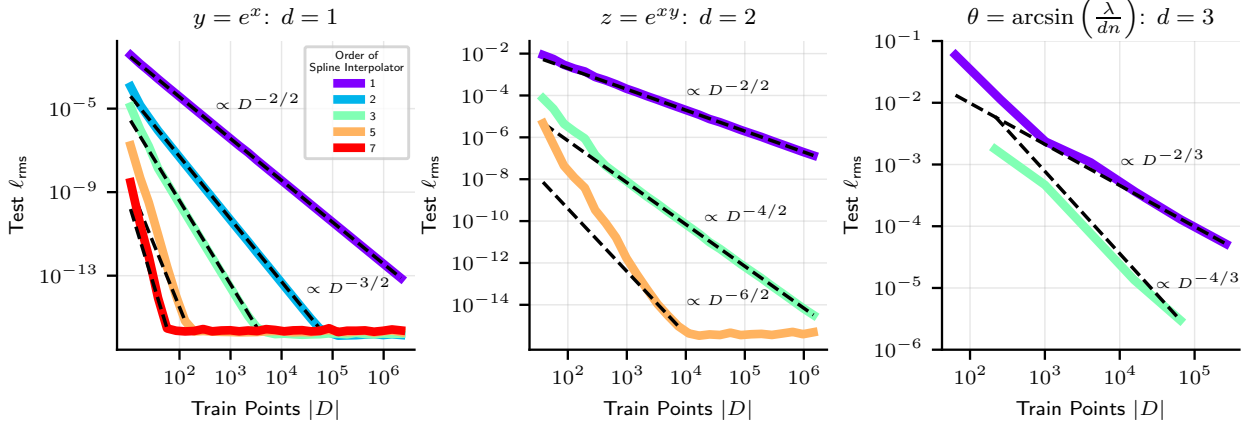


Figure 4.4: Interpolation methods, both linear and nonlinear, on 2D and 3D problems, seen to approximately scale as $D^{-(n+1)/d}$ where n is the order of the polynomial spline, d is the input dimension.

There are relatively few options for high-dimensional nonlinear interpolation.⁵ A particularly interesting parametrization of nonlinear functions is given by neural networks with nonlinear activation functions (and not piecewise linear like ReLUs). In Figure 4.5, we show how neural networks with tanh activations scale in increasing width. We observe that on some problems, they do better than the ideal scaling achievable with linear methods (shown as a green dashed line). However, in our experiments, they can sometimes scale worse, perhaps the result of imperfect optimization. Also, we find that scaling is typically not nearly as clean as a power law as it was for ReLU networks.

For some problems, one can show theoretically that architecture error can be made arbitrarily low, and that the loss is due entirely to optimization error and the generalization gap. As shown in [36], a two-layer neural network with only four hidden units can perform multiplication between two real numbers, provided that a twice-differentiable activation function is used. See Figure 4.6(b) for a diagram of such a network, taken from [36]. Note that this network becomes more accurate in the limit that some of its parameters become very small and others become very large. This result, that small neural networks can express multiplication arbitrarily well, implies that neural network architecture error is effectively zero for some problems. However, actually *learning* this multiplication circuit in practice is challenging since it involves some network parameters *diverging* $\rightarrow \infty$ while others $\rightarrow 0$ in a precise ratio. This means that for some tasks, neural network performance is mainly limited not by architecture error, but by optimization error.

Indeed, on some problems, a failure to achieve high precision can be blamed entirely on the optimization error. In Figure 4.6(a), we show neural network scaling on the equation $f(x_1, x_2, x_3, y_1, y_2, y_3) = x_1y_1 + x_2y_2 + x_3y_3$. For this problem, a 2-layer network with 12 hidden units (implementing three multiplications in parallel, with their results added in the last layer) can achieve ≈ 0 architecture error. Yet we see a failure to get anywhere near that architecture error or the noise floor set by machine precision. Instead, as one scales up the network size on this task, we see that despite architecture error abruptly dropping to zero early on, the actually attained loss continues to scales down smoothly.

⁵One method, which we have not tested, is radial basis function interpolation.

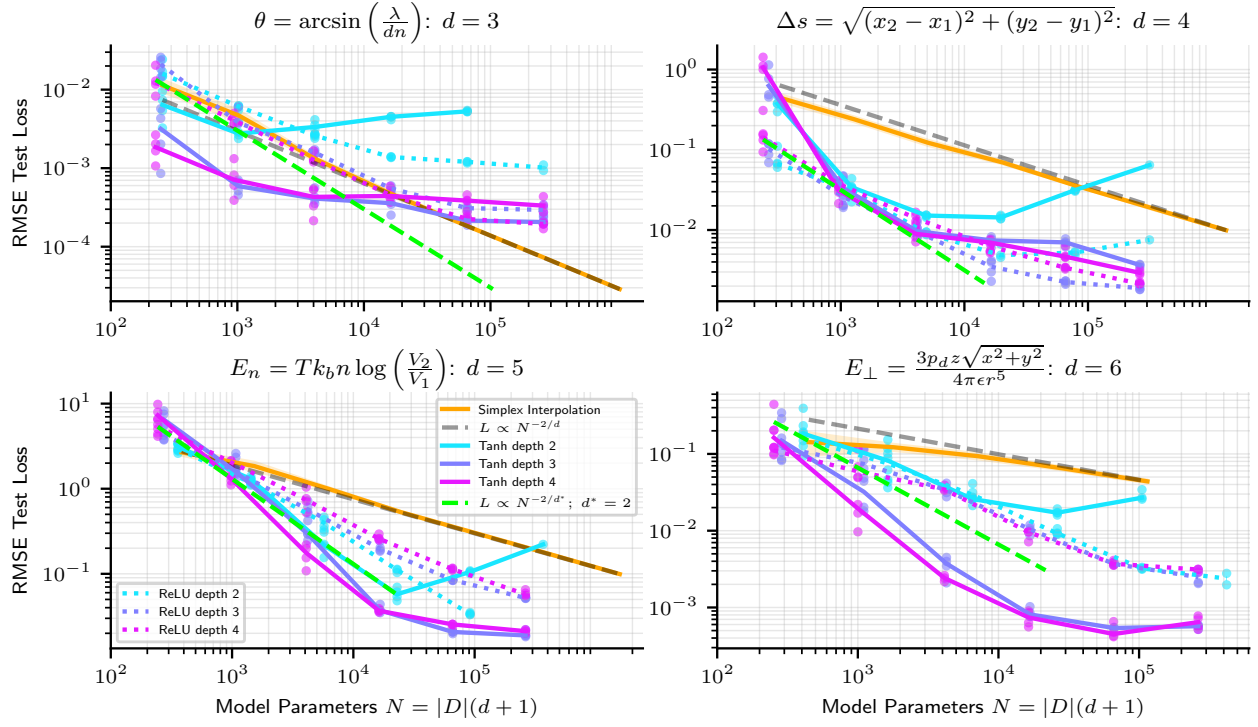


Figure 4.5: Scaling of linear simplex interpolation vs tanh NNs. We also plot ReLU NN performance as a dotted line for comparison. While simplex interpolation scales very predictably as $N^{-2/d}$, where d is the input dimension, tanh NN scaling is much messier.

It is therefore important to analyze the problem of optimization for high precision, which we do in the next section.

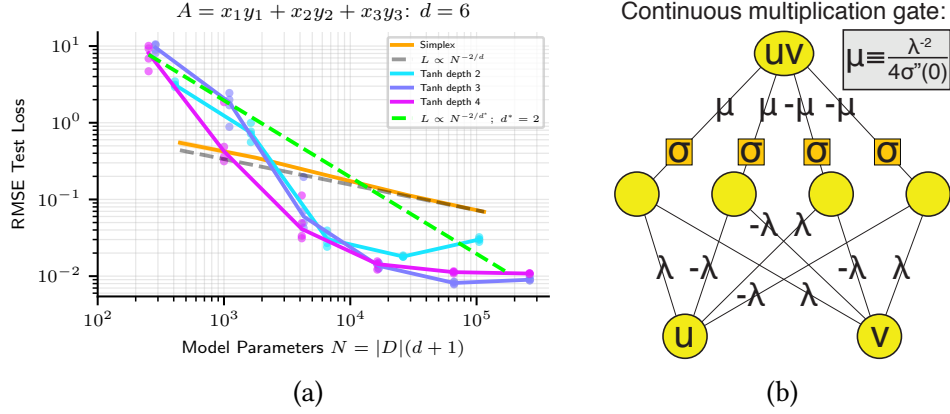


Figure 4.6: (a) Scaling of neural networks on a target function which can be arbitrarily closely approximated by a network of finite width. (b) diagram from [36] showing how a 4-neuron network can implement multiplication arbitrarily well. Therefore a depth-2 network of width at least 12 has an architecture error at the machine precision limit, yet optimization in practice does not discover solutions within at least 10 orders of magnitude of the precision limit.

4.4 Optimization

As seen above, when deep neural networks are trained with standard optimizers, they can produce significant optimization error, i.e. fail to find the best approximation. In this section, we discuss the difficulty of optimization in the high-precision regime and explore a few tricks for improving neural network training.

4.4.1 Properties of Loss Landscape

To understand the difficulty of optimizing in the high-precision regime, we attempt to understand the local geometry of the loss landscape at low loss. In particular, we compute the Hessian of the loss and study its eigenvalues. In Figure 4.7, we plot the spectrum of the Hessian, along with the magnitude of the gradient projected onto each of the corresponding eigenvectors, at a point in the loss landscape found by training with the Adam optimizer for 30k steps in a teacher-student setup. The teacher is a depth-3, width-3 tanh MLP and the student is a depth-3, width-40 tanh MLP. In line with [37–39], we find that at low loss, the loss landscape has a top cluster of directions of high curvature (relatively large positive eigenvalues) and a bulk of directions of very low curvature. Furthermore, the gradient tends to point most strongly in directions of higher curvature, and has very little projection onto directions of low curvature magnitude.

The basic picture emerging from this analysis is that of a canyon, i.e., a very narrow, very long valley around a low-loss minimum. The valley has steep walls in high-curvature directions and a long basin in low-curvature directions. Further reducing loss in this environment requires either (1) taking very precisely-sized steps along high-curvature directions to find the exact middle of the canyon or (2) moving along the canyon in low-curvature directions instead, almost orthogonally to the gradient. In this landscape, typical first-order optimizers used in deep learning may struggle to do either of these things, except perhaps if learning rates are chosen extremely carefully.

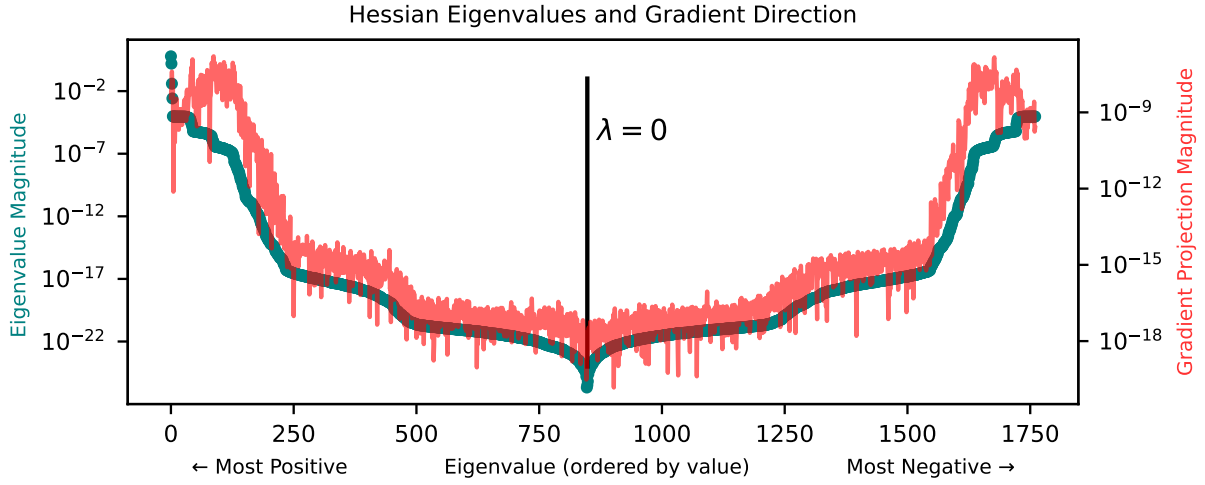


Figure 4.7: Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training with the Adam optimizer, along with the magnitude of the gradient’s projection onto each corresponding eigenvector (thin red line). We see a cluster of top eigenvalues and a bulk of near-zero eigenvalues. The gradient (thin jagged red curve) points mostly in directions of high-curvature.

4.4.2 Optimization Tricks For Reducing Optimization Error

How can we successfully optimize in such a poorly-conditioned, low-loss regime? We first find that switching from first-order optimizers like Adam to second-order optimizers like BFGS [40] can improve RMSE loss by multiple orders of magnitude. Second-order methods often both (1) employ line searches, and (2) search in directions not strongly aligned with the gradient, allowing optimization to progress within low-curvature subspaces. However, methods like BFGS are eventually bottlenecked by numerical precision limitations. To further lower loss, we tested the following two methods:

Low-Curvature Subspace Optimization We find that by restricting our optimization to low-curvature subspaces, we can further decrease loss past the point where loss of precision prevented BFGS from taking further steps. Our method has a single hyperparameter τ . The method is as follows: let $g = \nabla_{\theta} \mathcal{L}$ be the gradient and H be the Hessian of the loss. Denote an eigenvector-eigenvalue pair of H by (e_i, λ_i) . Instead of stepping in the direction $-g$, we instead compute $\hat{g} = \sum_{i: \lambda_i < \tau} e_i (e_i \cdot g)$. Essentially, we just project g onto the subspace spanned by eigenvalues of e_i such that $\lambda_i < \tau$. We then perform a line search to minimize loss along the direction $-\hat{g}$, and repeat the process. Note that this requires computing eigenvectors and eigenvalues for the whole Hessian H .

Boosting: staged training of neural networks We also investigated techniques related to the common practice of “boosting” in ML [41–44]. We found the following method to work quite well in our setting. Instead of training a full network to fit the target f , one can train two networks $f_{\theta_1}^{(1)}, f_{\theta_2}^{(2)}$ sequentially: first train $f_{\theta_1}^{(1)}$ to fit f , then train $f_{\theta_2}^{(2)}$ to fit the residual $\frac{f - f_{\theta_1}^{(1)}}{c}$, where $c \ll 1$

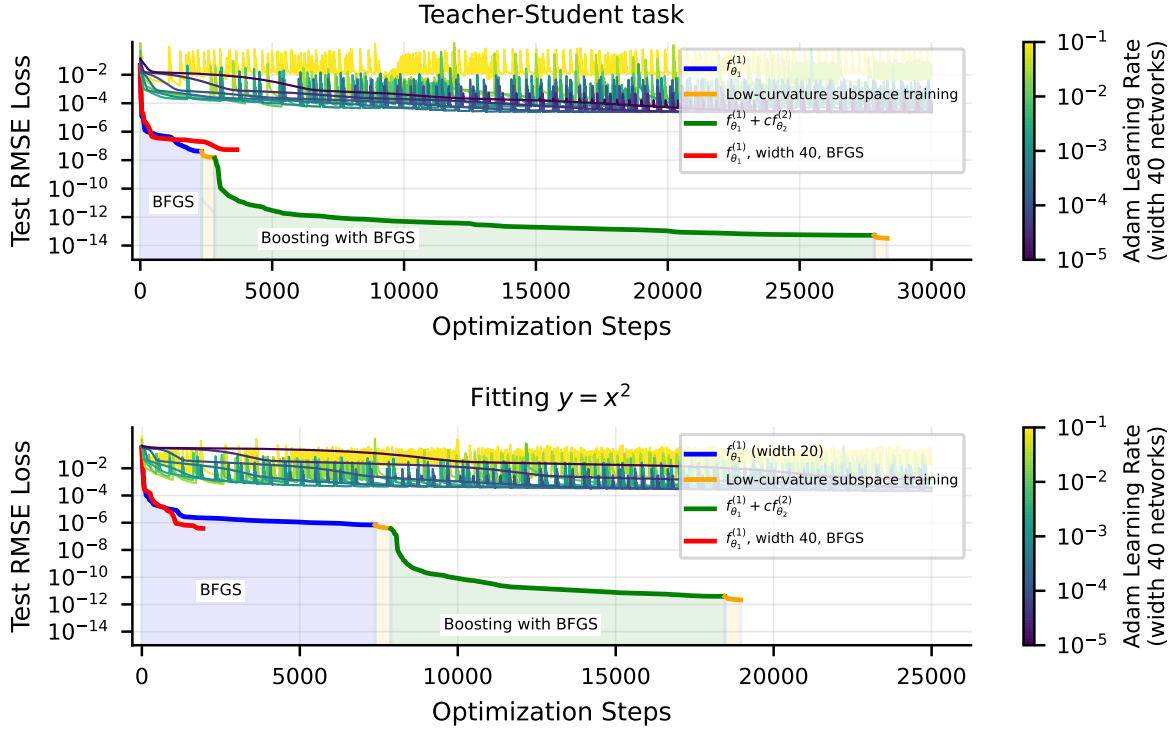


Figure 4.8: Comparison of Adam with BFGS + low-curvature subspace training + boosting. Using second-order methods like BFGS, but especially using boosting, leads to an improvement of many orders of magnitude over just training with Adam. Target functions are a teacher network (**top**) and a symbolic equation (**bottom**).

normalizes the residual to be of order unity. One can then combine the two networks into a single model $f(x) \approx f_{\theta_1}^{(1)}(x) + cf_{\theta_2}^{(2)}(x)$. If networks $f_{\theta_1}^{(1)}, f_{\theta_2}^{(2)}$ have widths w_1, w_2 respectively, then they can be combined into one network of width w_1, w_2 , with block-diagonal weight matrices, and where the parameters of the last layer of $f_{\theta_2}^{(2)}$ are scaled down by c . While further boosting steps can be performed, we found that there are quickly diminishing returns to training on successive residuals beyond that of the first network.

We find that, for low-dimensional problems, we can achieve substantially lower loss with these techniques. We use the following setup: we train width-40 depth-3 tanh MLPs to fit single-variable polynomials with the BFGS optimizer on MSE loss. The SciPy [27] BFGS implementation achieves 10^{-7} RMSE loss before precision loss prevents further iterations. Subsequently using low-curvature subspace training with a threshold $\tau = 10^{-16}$ can further lower RMSE loss a factor of over 2x. On similar low-dimensional problems, as shown in Figure 4.8, applying boosting, training a second network with BFGS on the residual of the first can lower RMSE loss further by 5-6 orders of magnitude. In Figure 4.8, we compare training runs with these tricks to runs with the Adam optimizer for a variety of learning rates. For our Adam training runs, we use width-40 tanh MLPs. When training with boosting, we train a width-20 network for $f_{\theta_1}^{(1)}$ and a width-20 network for $f_{\theta_2}^{(2)}$, for a combined width of 40. We also plot a width-40 network trained solely with BFGS

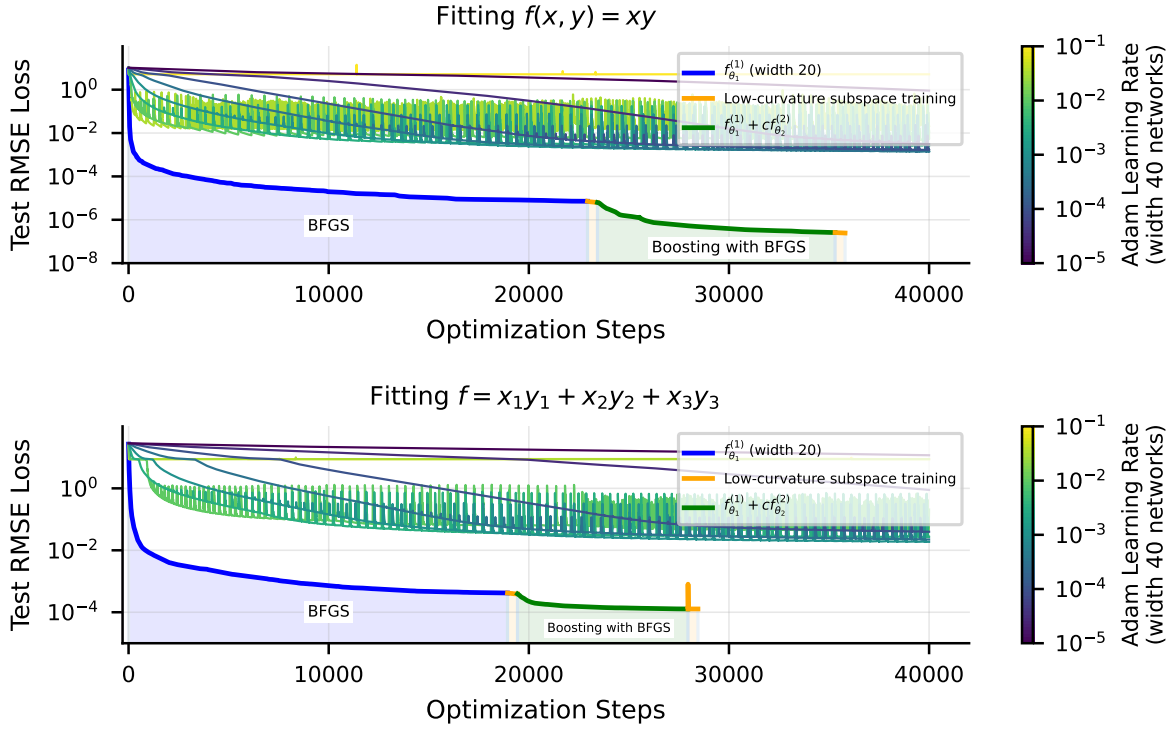


Figure 4.9: Comparison of Adam with BFGS + low-curvature subspace training + boosting, for a 2D problem (top) and a 6D problem (bottom), the equation we studied in Figure 4.6(a). As we increase dimension, the optimization tricks we tried in this work show diminishing benefits.

for comparison. We find, unsurprisingly, that BFGS significantly outperforms Adam. With our tricks, particularly boosting, we can sometimes outperform even well-tuned Adam by 8 orders of magnitude, driving RMSE loss down to $\approx 10^{-14}$, close to the machine precision limit.

4.4.3 Limitations and Outlook

The techniques we described above are not a silver bullet for fitting neural networks to any data with high precision. Firstly, second-order optimizers like BFGS scale poorly with the number of model parameters N (since the Hessian is an $N \times N$ matrix), limiting their applicability to small models. Also, we find that the gains from boosting diminish quickly as the input dimension of the problem grows. In Figure 4.9, we see that on the 6-dimensional problem discussed earlier (Figure 4.6(a)), BFGS + boosting achieves only about a 2-order of magnitude improvement, bringing the RMSE loss from 10^{-2} to 10^{-4} .

While boosting does not help much for high-dimensional problems, its success on low-dimensional problems is still noteworthy. By training two parts of a neural network separately and sequentially, we were able to dramatically improve performance. This suggests that perhaps there are other methods, not yet explored, for training and assembling neural networks in nonstandard ways to achieve dramatically better precision. The solutions found with boosting, where some network weights are at a much smaller scale than others, are not likely to be found with typical

training. An interesting avenue for future work would be exploring new initialization schemes, or other ways of training networks sequentially, to discover better solutions in underexplored regions of parameter space.

4.5 Conclusions

We have studied the problem of fitting scientific data with a variety of approximation methods, analyzing sources of error and their scaling.

- **Linear Simplex Interpolation** provides a piecewise linear fit to data, with RMSE loss scaling reliably as $D^{-2/d}$. Linear simplex interpolation always fits the training points exactly, and so error comes from the generalization gap and the architecture error.
- **ReLU Neural Networks** also provide a piecewise linear fit to data. Their performance (RMSE loss) often scales as D^{-2/d^*} , where d^* is the *maximum arity* of the task (typically $d^* = 2$). Accordingly, they can scale better than linear simplex interpolation when $d > 2$. Unfortunately, they are often afflicted by optimization error making them scale worse than linear simplex interpolation on 1D and 2D problems, and even in higher dimensions in the large-network limit.
- **Nonlinear Splines** approximate a target function piecewise by polynomials. They scale as $D^{-(n+1)/d}$ where n is the order of the polynomial.
- **Neural Networks with smooth activations** provide a nonlinear fit to data. Quite small networks with twice-differentiable nonlinearities can perform multiplication arbitrarily well [36], and so for many of the tasks we study (given by symbolic formulas), their architecture error is zero. We find that their inaccuracy does not appear to scale cleanly as power-laws. Optimization error is unfortunately a key driver of the error of these methods, but with special training tricks, we found that we could reduce RMSE loss on 1D problems down within 2-4 orders of magnitude of the 64-bit machine precision limit $\epsilon_0 \sim 10^{-16}$.

For those seeking high-precision fits, these results suggest the following heuristics, summarized in Figure 4.10 as a “User’s Guide to Precision”: If data dimensionality d is low ($d \leq 2$), polynomial spline interpolation can provide a fit at machine precision if you (1) have enough data and (2) choose a high enough polynomial order. Neural networks with smooth activations may in some cases also approach machine precision, possibly with less data, if they are trained with second-order optimizers like BFGS and boosted. For higher-dimensional problems ($d \geq 3$), neural networks are typically the most promising choice, since they can learn compositional modular structure that allows them to scale *as if* the data dimensionality were lower.

In summary, our results highlight both advantages and disadvantages of using neural networks to fit scientific data. We hope that they will help provide useful building blocks for further work towards precision machine learning.

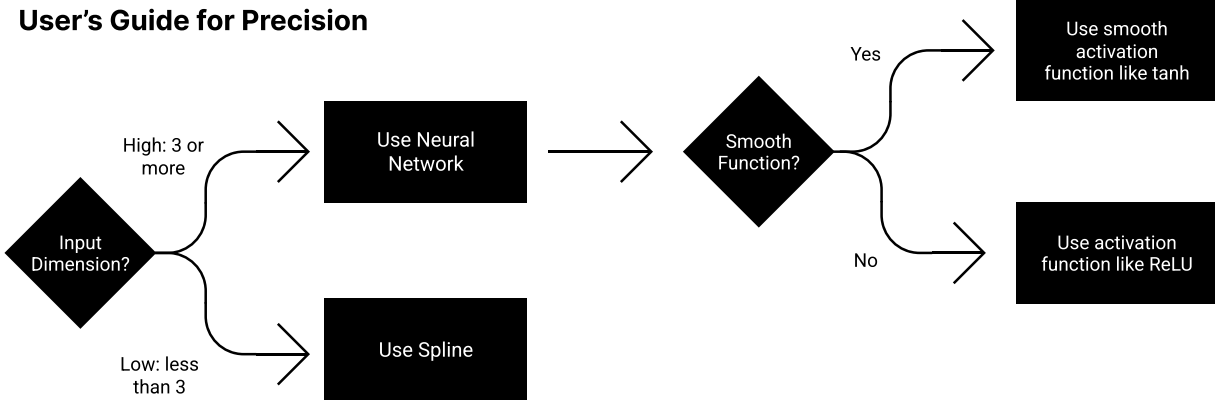


Figure 4.10: User's Guide for Precision: which approximation is best depends on properties of the problem.

Acknowledgements

This work was supported by The Casey Family Foundation, the Foundational Questions Institute, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), and IAIFI through NSF grant PHY-2019786.

References

- [1] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. "Deep Learning with Limited Numerical Precision." In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1737–1746. URL: <https://proceedings.mlr.press/v37/gupta15.html>.
- [2] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. "Mixed precision training." *arXiv preprint arXiv:1710.03740*, 2017.
- [3] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, et al. "A study of BFLOAT16 for deep learning training." *arXiv preprint arXiv:1905.12322*, 2019.
- [4] Y. Wang, C.-Y. Lai, J. Gómez-Serrano, and T. Buckmaster. *Asymptotic self-similar blow up profile for 3-D Euler via physics-informed neural networks*. 2022. DOI: [10.48550/ARXIV.2201.06780](https://arxiv.org/abs/2201.06780). URL: <https://arxiv.org/abs/2201.06780>.
- [5] V. Jejjala, D. K. M. Pena, and C. Mishra. "Neural network approximations for Calabi-Yau metrics." *Journal of High Energy Physics*, 2022(8), 2022, pp. 1–37.
- [6] J. Martyn, D. Luo, and K. Najafi. "Applying the Variational Principle to Quantum Field Theory with Neural-Networks." *Bulletin of the American Physical Society*, 2023.

- [7] D. Wu, L. Wang, and P. Zhang. “Solving statistical mechanics using variational autoregressive networks.” *Physical review letters*, 122(8), 2019, p. 080602.
- [8] S.-M. Udrescu and M. Tegmark. “AI Feynman: A physics-inspired method for symbolic regression.” *Science Advances*, 6(16), 2020, eaay2631.
- [9] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark. “AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity.” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 4860–4871.
- [10] S.-M. Udrescu and M. Tegmark. “AI Feynman: A physics-inspired method for symbolic regression.” *Science Advances*, 6(16), 2020, eaay2631.
- [11] R. B. Leighton and M. Sands. *The Feynman lectures on physics*. Addison-Wesley Boston, MA, USA, 1965.
- [12] “IEEE Standard for Floating-Point Arithmetic.” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019, pp. 1–84. DOI: [10.1109/IEEESTD.2019.8766229](https://doi.org/10.1109/IEEESTD.2019.8766229).
- [13] I. Gühring, M. Raslan, and G. Kutyniok. “Expressivity of deep neural networks.” *arXiv preprint arXiv:2007.04759*, 2020.
- [14] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. Patwary, M. Ali, Y. Yang, and Y. Zhou. “Deep learning scaling is predictable, empirically.” *arXiv preprint arXiv:1712.00409*, 2017.
- [15] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling laws for neural language models.” *arXiv preprint arXiv:2001.08361*, 2020.
- [16] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T. B. Brown, P. Dhariwal, S. Gray, et al. “Scaling laws for autoregressive generative modeling.” *arXiv preprint arXiv:2010.14701*, 2020.
- [17] D. Hernandez, J. Kaplan, T. Henighan, and S. McCandlish. “Scaling laws for transfer.” *arXiv preprint arXiv:2102.01293*, 2021.
- [18] B. Ghorbani, O. Firat, M. Freitag, A. Bapna, M. Krikun, X. Garcia, C. Chelba, and C. Cherry. “Scaling laws for neural machine translation.” *arXiv preprint arXiv:2109.07740*, 2021.
- [19] M. A. Gordon, K. Duh, and J. Kaplan. “Data and parameter scaling laws for neural machine translation.” In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 2021, pp. 5915–5922.
- [20] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer. “Scaling vision transformers.” In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12104–12113.
- [21] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. “Training Compute-Optimal Large Language Models.” *arXiv preprint arXiv:2203.15556*, 2022.
- [22] A. Clark, D. de Las Casas, A. Guy, A. Mensch, M. Paganini, J. Hoffmann, B. Damoc, B. Hechtman, T. Cai, S. Borgeaud, et al. “Unified scaling laws for routed language models.” In: *International Conference on Machine Learning*. PMLR. 2022, pp. 4057–4086.

- [23] U. Sharma and J. Kaplan. “A neural scaling law from the dimension of the data manifold.” *arXiv preprint arXiv:2004.10802*, 2020.
- [24] Y. Bahri, E. Dyer, J. Kaplan, J. Lee, and U. Sharma. “Explaining neural scaling laws.” *arXiv preprint arXiv:2102.06701*, 2021.
- [25] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee. “Understanding deep neural networks with rectified linear units.” *arXiv preprint arXiv:1611.01491*, 2016.
- [26] K. Eckle and J. Schmidt-Hieber. “A comparison of deep networks with ReLU activation function and linear spline-type methods.” *Neural Networks*, **110**, 2019, pp. 232–242.
- [27] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python.” *Nature methods*, **17**(3), 2020, pp. 261–272.
- [28] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization.” *arXiv preprint arXiv:1412.6980*, 2014.
- [29] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. “Efficient backprop.” In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.
- [30] T. Poggio. *Compositional Sparsity: a framework for ML*. Tech. rep. Center for Brains, Minds and Machines (CBMM), 2022.
- [31] W. Dahmen. “Compositional Sparsity, Approximation Classes, and Parametric Transport Equations.” *arXiv preprint arXiv:2207.06128*, 2022.
- [32] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao. “Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review.” *International Journal of Automation and Computing*, **14**(5), 2017, pp. 503–519.
- [33] M. Kohler and S. Langer. “On the rate of convergence of fully connected deep neural network regression estimates.” *The Annals of Statistics*, **49**(4), 2021, pp. 2231–2249.
- [34] B. Bauer and M. Kohler. “On deep learning as a remedy for the curse of dimensionality in nonparametric regression.” *The Annals of Statistics*, **47**(4), 2019, pp. 2261–2285.
- [35] F. Lekien and J. Marsden. “Tricubic interpolation in three dimensions.” *International Journal for Numerical Methods in Engineering*, **63**(3), 2005, pp. 455–471.
- [36] H. W. Lin, M. Tegmark, and D. Rolnick. “Why does deep and cheap learning work so well?” *Journal of Statistical Physics*, **168**(6), 2017, pp. 1223–1247.
- [37] L. Sagun, L. Bottou, and Y. LeCun. “Eigenvalues of the hessian in deep learning: Singularity and beyond.” *arXiv preprint arXiv:1611.07476*, 2016.
- [38] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou. “Empirical analysis of the hessian of over-parametrized neural networks.” *arXiv preprint arXiv:1706.04454*, 2017.
- [39] G. Gur-Ari, D. A. Roberts, and E. Dyer. “Gradient descent happens in a tiny subspace.” *arXiv preprint arXiv:1812.04754*, 2018.
- [40] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer, 1999.

- [41] T. Chen and C. Guestrin. “Xgboost: A scalable tree boosting system.” In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.
- [42] H. Schwenk and Y. Bengio. “Training methods for adaptive boosting of neural networks.” *Advances in neural information processing systems*, **10**, 1997.
- [43] H. Schwenk and Y. Bengio. “Boosting neural networks.” *Neural computation*, **12**(8), 2000, pp. 1869–1887.
- [44] S. Badirli, X. Liu, Z. Xing, A. Bhowmik, K. Doan, and S. S. Keerthi. “Gradient boosting neural networks: Grownnet.” *arXiv preprint arXiv:2002.07971*, 2020.

4.6 Appendix

4.6.1 Boosting advantage comes not just from rescaling loss

When one trains a second network on the residual of a first trained network during boosting, the scale of the loss, at least early in training, lies around order unity, much higher than the loss at the end of training the first network. Does the advantage of boosting then lie in increasing the value of the loss to avoid precision limits within BFGS? To investigate this, instead of performing boosting, we instead simply rescale the loss after training width-40 networks with BFGS and further train the same network with BFGS. In Figure 4.11, we plot training curves for varying amounts of rescaling of the loss function. We find that the benefits of rescaling the loss are small compared with boosting. At best, we achieved an $\approx 4x$ reduction in RMSE loss, compared with the 10000-100000x reduction from boosting.

4.6.2 A closer look at low-curvature subspace optimization

Low-curvature subspace optimization tends to reduce loss much less than boosting; it is difficult to even see improvement from low-curvature subspace training in Figures 4.8 and 4.9. In Figure 4.12, we show training curves for low-curvature subspace optimization for a width-20 depth-3 tanh network after being trained with BFGS. We see that for appropriately chosen curvature thresholds τ , we can reduce RMSE loss by 2-3x, although the method is admittedly expensive, requiring a full computation of the Hessian.

4.6.3 Neural scaling of tanh networks with BFGS versus Adam optimizer

In Section 4.3, we found that tanh networks sometimes scaled poorly, and suggested that this was due to challenges with optimization rather than the expressivity of the architecture. In Figure 4.13, we compare neural scaling curves when training with Adam versus the BFGS optimizer for the same problems as Figure 4.5.

4.6.4 Loss landscape at lower loss

In Figure 4.7, we plotted the Hessian eigenvalues at the end of training with the Adam optimizer. In Figure 4.14, we do the same but for a point in the loss landscape found by the BFGS optimizer

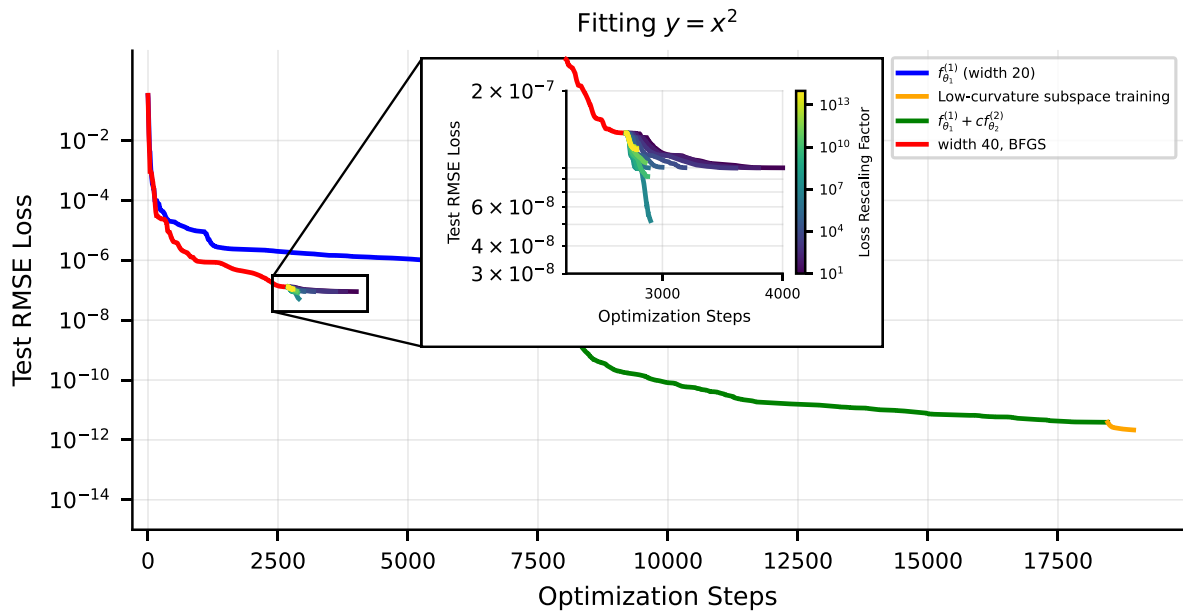


Figure 4.11: Rescaling loss has minimal benefit relative to boosting.

rather than Adam. The basic picture of a “canyon” discussed in Section 4.4 is clearer at lower-loss points in the landscape found by BFGS – there is a clear set of high-curvature directions which the gradient mostly points along.

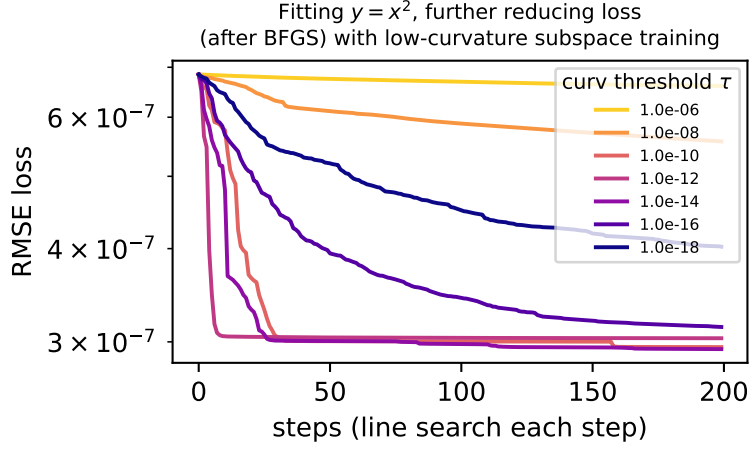


Figure 4.12: Low-curvature subspace training curves for varying thresholds τ .

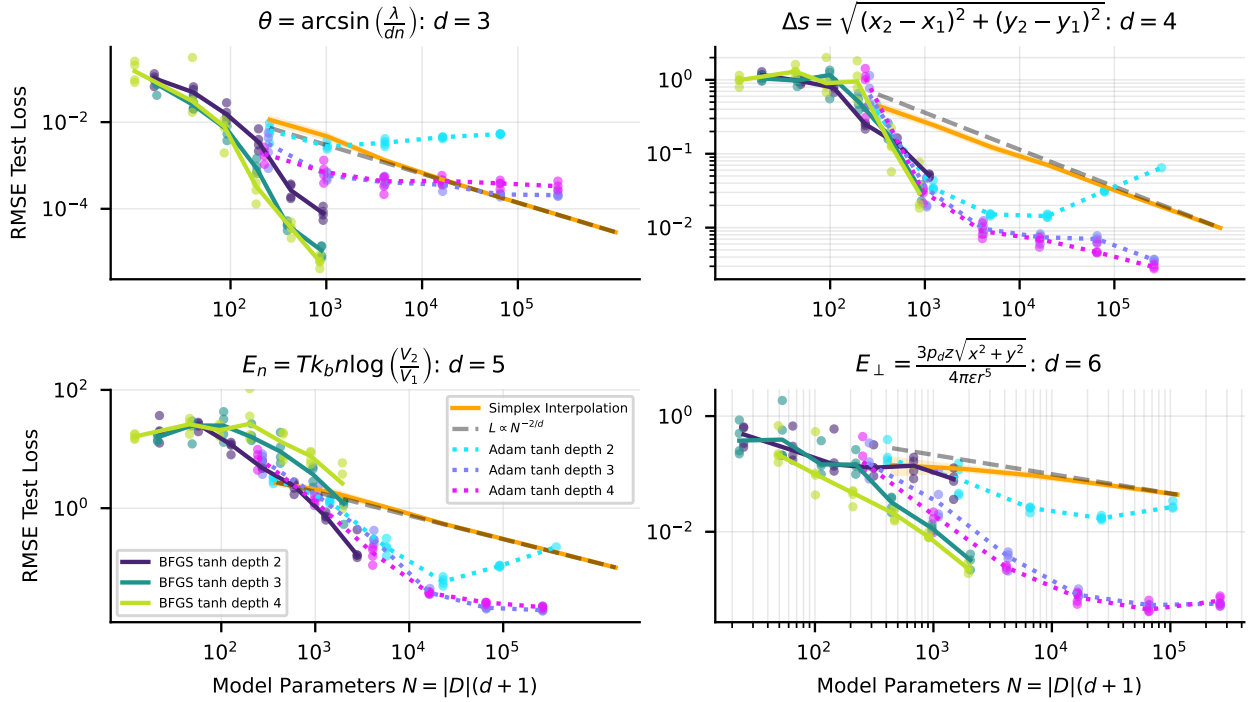


Figure 4.13: Scaling of tanh networks with the BFGS vs Adam optimizer. We use the same setup as in Figure 4.5, training tanh MLPs of depth 2-4 of varying width on functions given by symbolic equations. BFGS outperforms Adam on the 3-dimensional example shown (top left) and performs roughly similarly to Adam on the other problems.

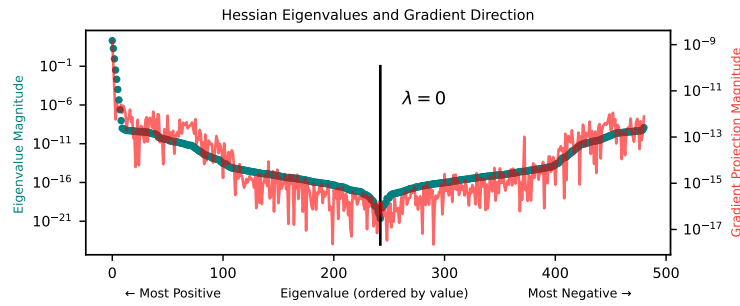


Figure 4.14: Eigenvalues (dark green) of the loss landscape Hessian (MSE loss) after training a width-20, depth-3 network to fit $y = x^2$ with the BFGS optimizer. Like in Figure 4.7, we also plot the magnitude of the gradient’s projection onto each corresponding eigenvector (thin red line). The “canyon” shape of the loss landscape is more obvious at lower-loss points in the landscape found by BFGS than Adam finds. There is a clear set of top eigenvalues corresponding to a few directions of much higher curvature than the bulk.

Chapter 5

Towards Understanding Grokking: An Effective Theory of Representation Learning^{*}

In this paper, we developed a theory of “grokking”. Grokking a phenomenon where neural networks first memorize their training set, but then eventually generalize with continued training. We found that during training, networks form highly structured representations of their inputs. We hypothesized that grokking occurs because these very particular representations, which presumably are necessary for generalization, take longer to form than the circuitry that memorizes the training set points. See Section 6.8.5 for commentary on how this work on grokking relates to other theories from the literature and the core ideas of this thesis.

Abstract

We aim to understand *grokking*, a phenomenon where models generalize long after overfitting their training set. We present both a *microscopic* analysis anchored by an effective theory and a *macroscopic* analysis of phase diagrams describing learning performance across hyperparameters. We find that generalization originates from structured representations whose training dynamics and dependence on training set size can be predicted by our effective theory in a toy setting. We observe empirically the presence of four learning phases: *comprehension*, *grokking*, *memorization*, and *confusion*. We find representation learning to occur only in a “Goldilocks zone” (including comprehension and grokking) between memorization and confusion. Compared to the comprehension phase, the grokking phase stays closer to the memorization phase, leading to delayed generalization. The Goldilocks phase is reminiscent of “intelligence from starvation” in Darwinian evolution, where resource limitations drive discovery of more efficient solutions. This study not only provides intuitive explanations of the origin of grokking, but also highlights the usefulness of physics-inspired tools, e.g., effective theories and phase diagrams, for understanding deep

^{*}Ziming Liu, Ouail Kitouni, Niklas S. Nolte, Eric J. Michaud, Max Tegmark, and Mike Williams. “Towards understanding grokking: An effective theory of representation learning.” *Advances in Neural Information Processing Systems* 35 (2022): 34651–34663.

learning.

5.1 Introduction

Perhaps *the* central challenge of a scientific understanding of deep learning lies in accounting for neural network generalization. Power et al. [1] recently added a new puzzle to the task of understanding generalization with their discovery of *grokking*. Grokking refers to the surprising phenomenon of *delayed generalization* where neural networks, on certain learning problems, generalize long after overfitting their training set. It is a rare albeit striking phenomenon that violates common machine learning intuitions, raising three key puzzles:

- Q1 *The origin of generalization*: When trained on the algorithmic datasets where grokking occurs, how do models generalize at all?
- Q2 *The critical training size*: Why does the training time needed to “grok” (generalize) diverge as the training set size decreases toward a critical point?
- Q3 *Delayed generalization*: Under what conditions does delayed generalization occur?

We provide evidence that representation learning is central to answering each of these questions. Our answers can be summarized as follows:

- A1 Generalization can be attributed to learning a good representation of the inputs, i.e. a representation that has the appropriate structure for the task. See Figures 5.1 and 5.2.
- A2 The critical training set size corresponds to the least amount of training data that can determine such a representation (which, in some cases, is unique up to linear transformations).
- A3 Grokking is a phase between “comprehension” and “memorization” phases and it can be remedied with proper hyperparameter tuning, as illustrated by the phase diagrams in Figure 5.5.

This paper is organized as follows: In Section 5.2, we introduce the problem setting and build a simplified toy model. In Section 5.3, we will use an *effective theory* approach, a useful tool from theoretical physics, to shed some light on questions Q1 and Q2 and show the relationship between generalization and the learning of structured representations. In Section 5.4, we explain Q3 by displaying phase diagrams from a grid search of hyperparameters and show how we can “de-delay” generalization by following intuition developed from the phase diagram. We discuss related work in Section 5.5, followed by conclusions in Section 5.6.

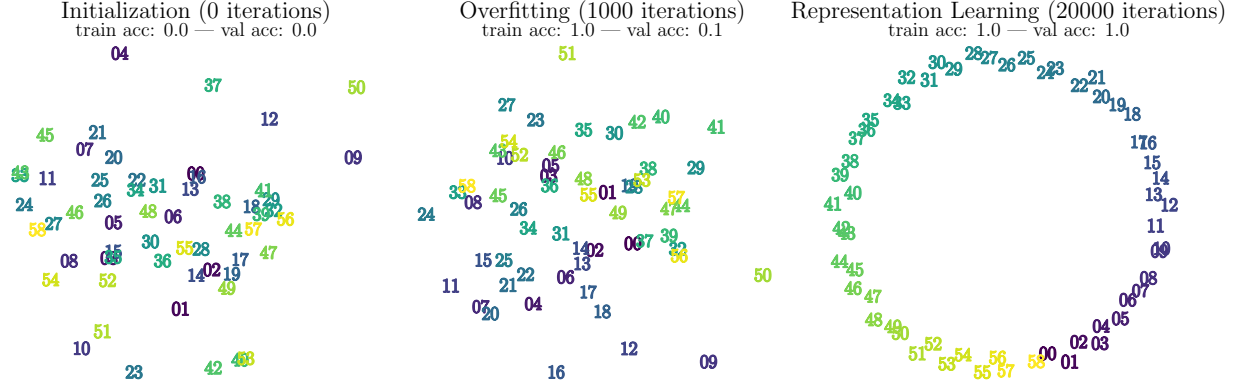


Figure 5.1: Visualization of the first two principal components of the learned input embeddings at different training stages of a transformer learning modular addition. We observe that generalization coincides with the emergence of structure in the embeddings. See Section 5.4.2 for the training details.

5.2 Problem Setting

Power et al. [1] observe grokking on an unusual task – learning an “algorithmic” binary operation. Given some binary operation $\circ$, a network is tasked with learning the map $(a, b) \mapsto c$ where $a \circ b = c$. They use a decoder-only transformer to predict the second to last token in a tokenized equation of the form “<lhs> <op> <rhs> <eq> <result> <eos>”. Each token is represented as a 256-dimensional embedding vector. The embeddings are learnable and initialized randomly. After the transformer, a final linear layer maps the output to class logits for each token.

Toy Model We primarily study grokking in a simpler toy model, which still retains the key behaviors from the setup of [1]. Although [1] treated this as a classification task, we study both regression and classification problems. The basic setup is as follows: our model takes as input the symbols a, b and maps them to trainable embedding vectors $\mathbf{E}_a, \mathbf{E}_b \in \mathbb{R}^{d_{\text{in}}}$. It then sums $\mathbf{E}_a, \mathbf{E}_b$ and sends the resulting vector through a “decoder” MLP. The target output vector, denoted $\mathbf{Y}_c \in \mathbb{R}^{d_{\text{out}}}$ is a fixed random vector (regression task) or a one-hot vector (classification task). Our model architecture can therefore be compactly described as $(a, b) \mapsto \text{Dec}(\mathbf{E}_a + \mathbf{E}_b)$, where the embeddings $\mathbf{E}_*$ and the decoder are trainable. Despite its simplicity, this toy model can generalize to all abelian groups (discussed in Appendix 5.7.1). In sections 5.3-5.4.1, we consider only the binary operation of addition. We consider modular addition in Section 5.4.2 to generalize some of our results to a transformer architecture and study general non-abelian operations in Appendix 5.7.6.

Dataset In our toy setting, we are concerned with learning the addition operation. A data sample corresponding to $i + j$ is denoted as (i, j) for simplicity. If $i, j \in \{0, \dots, p-1\}$, there are in total $p(p+1)/2$ different samples since we consider $i + j$ and $j + i$ to be the same sample. A dataset D is a set of non-repeating data samples. We denote the full dataset as D_0 and split it into a training dataset D and a validation dataset D' , i.e., $D \cup D' = D_0$, $D \cap D' = \emptyset$. We define *training data fraction* = $|D|/|D_0|$ where $|\cdot|$ denotes the cardinality of the set.

5.3 Why Generalization: Representations and Dynamics

We can see that generalization appears to be linked to the emergence of highly-structured embeddings in Figure 5.2. In particular, Figure 5.2 (b) shows parallelograms in toy addition, and (d) shows a circle in toy modular addition. We now restrict ourselves to the toy addition setup and formalize a notion of representation quality and show that it predicts the model’s performance. We then develop a physics-inspired *effective* theory of learning which can accurately predict the critical training set size and training trajectories of representations. In physics, effective theories aim to describe observed phenomena without implying that the mechanisms used in the description necessarily have an exact counterpart in the true underlying causes. In our effective theory, we will model the dynamics of representation learning not as gradient descent of the true task loss but rather a simpler effective loss function ℓ_{eff} which depends only on the representations and not on the decoder.

5.3.1 Representation quality predicts generalization for the toy model

A rigorous definition for *structure* in the learned representation is necessary. We propose the following definition,

Definition 1. (i, j, m, n) is a δ -parallelogram in the representation $\mathbf{R} \equiv [\mathbf{E}_0, \dots, \mathbf{E}_{p-1}]$ if

$$|(\mathbf{E}_i + \mathbf{E}_j) - (\mathbf{E}_m + \mathbf{E}_n)| \leq \delta.$$

In the following derivations, we can take δ , which is a small threshold to tolerate numerical errors, to be zero.

As long as training loss is effectively zero, any parallelogram (i, j, m, n) should satisfy $i + j = m + n$. Suppose that this is not the case, i.e., suppose $\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n$ but $i + j \neq m + n$, then $\mathbf{Y}_{i+j} = \text{Dec}(\mathbf{E}_i + \mathbf{E}_j) = \text{Dec}(\mathbf{E}_m + \mathbf{E}_n) = \mathbf{Y}_{m+n}$ where the first and last equalities come from the zero training loss assumption. However, since $i + j \neq m + n$, we have $\mathbf{Y}_{i+j} \neq \mathbf{Y}_{m+n}$ (almost surely in the regression task), a contradiction.

It is convenient to define the permissible parallelogram set associated with a training dataset D (“permissible” means consistent with 100% training accuracy) as

$$P_0(D) = \{(i, j, m, n) | (i, j) \in D, (m, n) \in D, i + j = m + n\}. \quad (5.1)$$

For simplicity, we denote $P_0 \equiv P_0(D_0)$. Given a representation $\mathbf{R}$, we can check how many permissible parallelograms actually exist in $\mathbf{R}$ within error δ , so we define the parallelogram set corresponding to $\mathbf{R}$ as

$$P(\mathbf{R}, \delta) = \{(i, j, m, n) | (i, j, m, n) \in P_0, |(\mathbf{E}_i + \mathbf{E}_j) - (\mathbf{E}_m + \mathbf{E}_n)| \leq \delta\}. \quad (5.2)$$

For brevity we will write $P(\mathbf{R})$, suppressing the dependence on δ . We define the representation quality index (RQI) as

$$\text{RQI}(\mathbf{R}) = \frac{|P(\mathbf{R})|}{|P_0|} \in [0, 1]. \quad (5.3)$$

We will use the term *linear representation* or *linear structure* to refer to a representation whose embeddings are of the form $\mathbf{E}_k = \mathbf{a} + k\mathbf{b}$ ($k = 0, \dots, p-1$; $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{d_{\text{in}}}$). A linear representation has

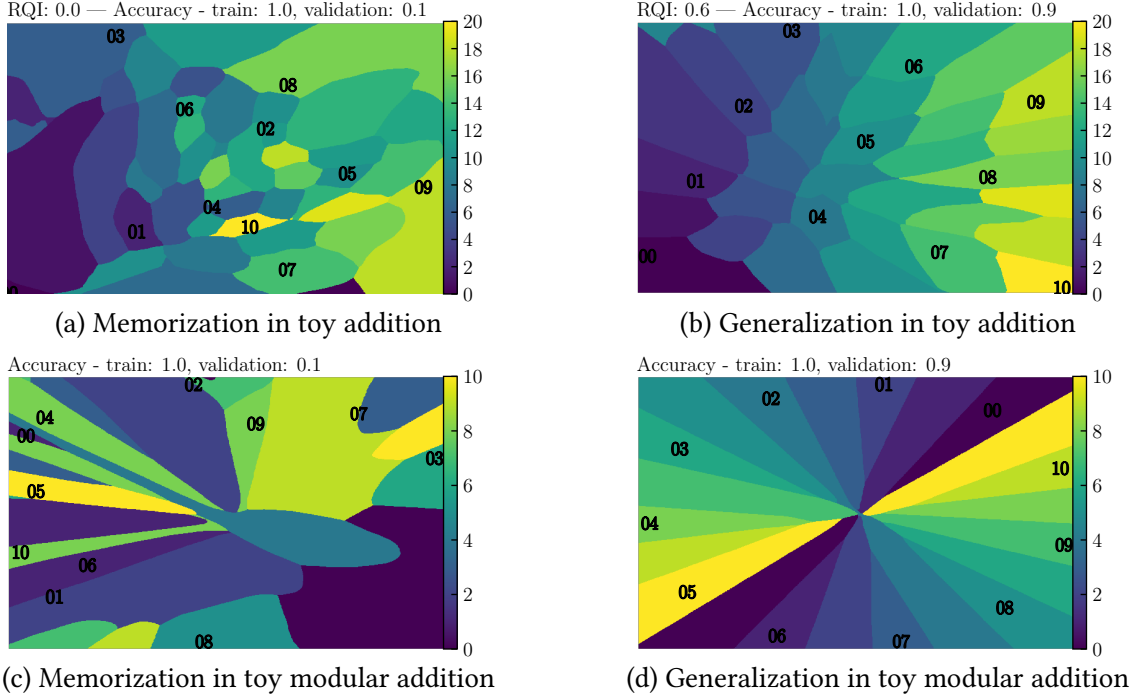


Figure 5.2: Visualization of the learned set of embeddings ($p = 11$) and the decoder function associated with it for the case of 2D embeddings. Axes refer to each dimension of the learned embeddings. The decoder is evaluated on a grid of points in embedding-space and the color at each point represents the highest probability class. For visualization purposes, the decoder is trained on inputs of the form $(\mathbf{E}_i + \mathbf{E}_j)/2$. One can read off the output of the decoder when fed the operation $i \circ j$ from this figure simply by taking the midpoint between the respective embeddings of i and j .

RQI = 1, while a random representation (sampled from, say, a normal distribution) has RQI = 0 with high probability.

Quantitatively, we denote the “predicted accuracy” $\widehat{\text{Acc}}$ as the accuracy achievable on the whole dataset given the representation $\mathbf{R}$ (see Appendix 5.7.2 for the full details). In Figure 5.3, we see that the predicted $\widehat{\text{Acc}}$ aligns well with the true accuracy Acc , establishing good evidence that representation leads to generalization. We use an example to illustrate the origin of generalization here. In the setup of Figure 5.2 (a), suppose the decoder can achieve zero training loss and $\mathbf{E}_6 + \mathbf{E}_8$ is a training sample hence $\text{Dec}(\mathbf{E}_6 + \mathbf{E}_8) = \mathbf{Y}_{14}$. At validation time, the decoder is tasked with predicting a validation sample $\mathbf{E}_5 + \mathbf{E}_9$. Since $(5, 9, 6, 8)$ forms a parallelogram such that $\mathbf{E}_5 + \mathbf{E}_9 = \mathbf{E}_6 + \mathbf{E}_8$, the decoder can predict the validation sample correctly because $\text{Dec}(\mathbf{E}_5 + \mathbf{E}_9) = \text{Dec}(\mathbf{E}_6 + \mathbf{E}_8) = \mathbf{Y}_{14}$.

5.3.2 The critical training set size

Suppose that we have an ideal model $\mathcal{M}^* = (\text{Dec}^*, \mathbf{R}^*)$ such that:¹

¹One can verify a posteriori if a trained model $\mathcal{M}$ is close to being an ideal model $\mathcal{M}^*$. Please refer to Appendix 5.7.3 for details.

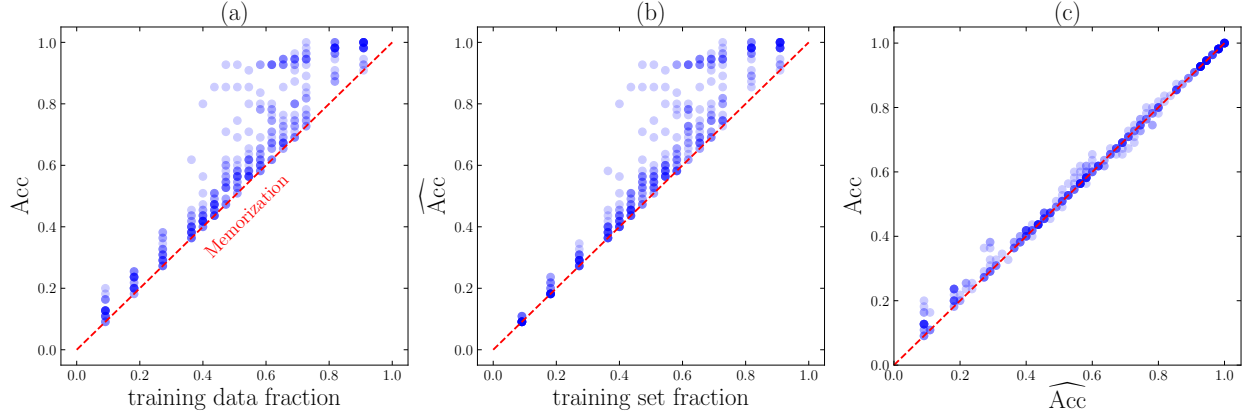


Figure 5.3: We compute accuracy (of the full dataset) either measured empirically Acc , or predicted from representation $\widehat{\text{Acc}}$. These two accuracies as a function of training data fraction are plotted in (a)(b), and their agreement is shown in (c).

- (1) $\mathcal{M}^*$ can achieve zero training loss;
- (2) $\mathcal{M}^*$ has an injective decoder, i.e., $\text{Dec}^*(\mathbf{x}_1) \neq \text{Dec}^*(\mathbf{x}_2)$ for any $\mathbf{x}_1 \neq \mathbf{x}_2$.

Then Proposition 1 provides a mechanism for the formation of parallelograms.

Proposition 1. If a training set D contains two samples (i, j) and (m, n) with $i + j = m + n$, then $\mathcal{M}^*$ learns a representation $\mathbf{R}^*$ such that $\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n$, i.e., (i, j, m, n) forms a parallelogram.

Proof. Due to the zero training loss assumption, we have $\text{Dec}^*(\mathbf{E}_i + \mathbf{E}_j) = \mathbf{Y}_{i+j} = \mathbf{Y}_{m+n} = \text{Dec}^*(\mathbf{E}_m + \mathbf{E}_n)$. Then the injectivity of Dec^* implies $\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n$. $\square$

The dynamics of the trained embedding parameters are determined by various factors interacting in complex ways, for instance: the details of the decoder architecture, the optimizer hyperparameters, and the various kinds of implicit regularization induced by the training procedure. We will see that the dynamics of normalized quantities, namely, the normalized embeddings at time t , defined as $\tilde{\mathbf{E}}_k^{(t)} = \frac{\mathbf{E}_k^{(t)} - \mu_t}{\sigma_t}$, where $\mu_t = \frac{1}{p} \sum_k \mathbf{E}_k^{(t)}$ and $\sigma_t = \frac{1}{p} \sum_k |\mathbf{E}_k^{(t)} - \mu_t|^2$, can be qualitatively described by a simple effective loss. We will assume that the normalized embedding vectors obey a gradient flow for an effective loss function of the form

$$\frac{d\tilde{\mathbf{E}}_i}{dt} = -\frac{\partial \ell_{\text{eff}}}{\partial \tilde{\mathbf{E}}_i}, \quad (5.4)$$

$$\ell_{\text{eff}} = \frac{\ell_0}{Z_0}, \quad \ell_0 \equiv \sum_{(i,j,m,n) \in P_0(D)} |\tilde{\mathbf{E}}_i + \tilde{\mathbf{E}}_j - \tilde{\mathbf{E}}_m - \tilde{\mathbf{E}}_n|^2, \quad Z_0 \equiv \sum_k |\tilde{\mathbf{E}}_k|^2, \quad (5.5)$$

where $|\cdot|$ denotes Euclidean vector norm. Note that the embeddings do not collapse to the trivial solution $\mathbf{E}_0 = \dots = \mathbf{E}_{p-1} = 0$ unless initialized as such, because two conserved quantities exist, as proven in Appendix 5.7.4:

$$\mathbf{C} = \sum_k \mathbf{E}_k, \quad Z_0 = \sum_k |\mathbf{E}_k|^2. \quad (5.6)$$

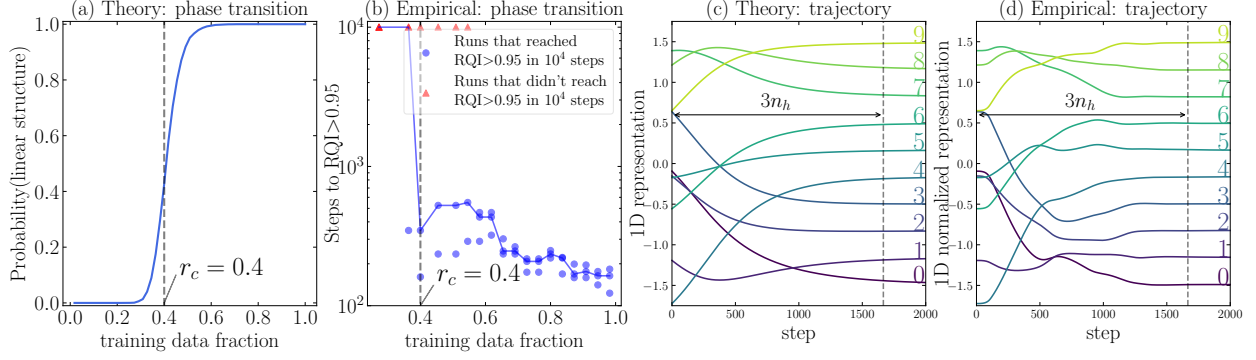


Figure 5.4: (a) The effective theory predicts a phase transition in the probability of obtaining a linear representation around $r_c = 0.4$. (b) Empirical results display a phase transition of RQI around $r_c = 0.4$, in agreement with the theory (the blue line shows the median of multiple random seeds). The evolution of 1D representations predicted by the effective theory or obtained from neural network training (shown in (c) and (d) respectively) agree creditably well.

We shall now use the effective dynamics to explain empirical observations such as the existence of a critical training set size for generalization.

Degeneracy of ground states (loss optima) We define ground states as those representations satisfying $\ell_{\text{eff}} = 0$, which requires the following linear equations to hold:

$$A(P) = \{\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n | (i, j, m, n) \in P\} \quad (5.7)$$

Since each embedding dimension obeys the same set of linear equations, we will assume, without loss of generality, that $d_{\text{in}} = 1$. The dimension of the null space of $A(P)$, denoted as n_0 , is the number of degrees of freedom of the ground states. Given a set of parallelograms implied by a training dataset D , the nullity of $A(P(D))$ could be obtained by computing the singular values $0 \leq \sigma_1 \leq \dots \leq \sigma_p$. We always have $n_0 \geq 2$, i.e., $\sigma_1 = \sigma_2 = 0$ because the nullity of $A(P_0)$, the set of linear equations given by all possible parallelograms, is $\text{Nullity}(A(P_0)) = 2$ which can be attributed to two degrees of freedom (translation and scaling). If $n_0 = 2$, the representation is unique up to translations and scaling factors, and the embeddings have the form $\mathbf{E}_k = \mathbf{a} + k\mathbf{b}$. Otherwise, when $n_0 > 2$, the representation is not constrained enough such that all the embeddings lie on a line.

We present theoretical predictions alongside empirical results for addition ($p = 10$) in Figure 5.4. As shown in Figure 5.4 (a), our effective theory predicts that the probability that the training set implies a unique linear structure (which would result in perfect generalization) depends on the training data fraction and has a phase transition around $r_c = 0.4$. Empirical results from training different models are shown in Figure 5.4(b). The number of steps to reach $\text{RQI} > 0.95$ is seen to have a phase transition at $r_c = 0.4$, agreeing with the proposed effective theory and with the empirical findings in [1].

Time towards the linear structure We define the Hessian matrix of ℓ_0 as

$$\mathbf{H}_{ij} = \frac{1}{Z_0} \frac{\partial^2 \ell_0}{\partial \mathbf{E}_i \partial \mathbf{E}_j}, \quad (5.8)$$

Note that $\ell_0 = \frac{1}{2} \mathbf{R}^T \mathbf{H} \mathbf{R}$, $\mathbf{R} = [\mathbf{E}_0, \mathbf{E}_1, \dots, \mathbf{E}_{p-1}]$, so the gradient descent is linear, i.e.,

$$\frac{d\mathbf{R}}{dt} = -\mathbf{H}\mathbf{R}. \quad (5.9)$$

If $\mathbf{H}$ has eigenvalues $\lambda_i = \sigma_i^2$ (sorted in increasing order) and eigenvectors $\bar{\mathbf{v}}_i$, and we have the initial condition $\mathbf{R}(t=0) = a_i \bar{\mathbf{v}}_i$, then we have $\mathbf{R}(t) = a_i \bar{\mathbf{v}}_i e^{-\lambda_i t}$. The first two eigenvalues vanish and $t_h = 1/\lambda_3$ determines the timescale for the slowest component to decrease by a factor of e . When the step size is η , the corresponding number of steps is $n_h = t_h/\eta = 1/(\lambda_3 \eta)$.

We verify the above analysis with empirical results. Figure 5.4 (c)(d) show the trajectories obtained from the effective theory and from neural network training, respectively. The 1D neural representation in Figure 5.4(d) are manually normalized to zero mean and unit variance. The two trajectories agree qualitatively, and it takes about $3n_h$ steps for two trajectories to converge to the linear structure. The quantitative differences might be due to the absence of the decoder in the effective theory, which assumes the decoder to be *adiabatic* in the sense that it takes infinitesimal step sizes.

Limitations of the effective theory While our theory defines an effective loss based on the Euclidean distance between embeddings $\mathbf{E}_i + \mathbf{E}_j$ and $\mathbf{E}_n + \mathbf{E}_m$, one could imagine generalizing the theory to define a broader notion of parallelogram given by some other metric on the representation space. For instance, if we have a decoder like in Figure 5.2(c) then the distance between distinct representations within the same “pizza slice” is low, meaning that representations arranged not in parallelograms w.r.t. the Euclidean metric may be parallelograms with respect to the metric defined by the decoder.

5.4 Delayed Generalization: A Phase Diagram

So far, we have (1) observed empirically that generalization on algorithmic datasets corresponds with the emergence of well-structured representations, (2) defined a notion of representation quality in a toy setting and shown that it predicts generalization, and (3) developed an effective theory to describe the learning dynamics of the representations in the same toy setting. We now study how optimizer hyperparameters affect high-level learning performance. In particular, we develop phase diagrams for how learning performance depend on the representation learning rate, decoder learning rate and the decoder weight decay. These parameters are of interest since they most explicitly regulate a kind of *competition* between the encoder and decoder, as we elaborate below.

5.4.1 Phase diagram of a toy model

Training details We update the representation and the decoder with different optimizers. For the 1D embeddings, we use the Adam optimizer with learning rate $[10^{-4}, 10^{-2}]$ and zero weight decay. For the decoder, we use an AdamW optimizer with the learning rate in $[10^{-4}, 10^{-2}]$ and the weight decay in $[0, 10]$ (regression) or $[0, 20]$ (classification). For training/validation splitting, we choose 45/10 for non-modulo addition ($p = 10$) and 24/12 for the permutation group S_3 . We hard-code addition or matrix multiplication (details in Appendix 5.7.6) in the decoder for the addition group and the permutation group, respectively.

For each choice of learning rate and weight decay, we compute the number of steps to reach high (90%) training/validation accuracy. The 2D plane is split into four phases: *comprehension*, *grokking*, *memorization* and *confusion*, defined in Table 5.1. Both comprehension and grokking are able to generalize (in the “Goldilocks zone”), although the grokking phase has delayed generalization.

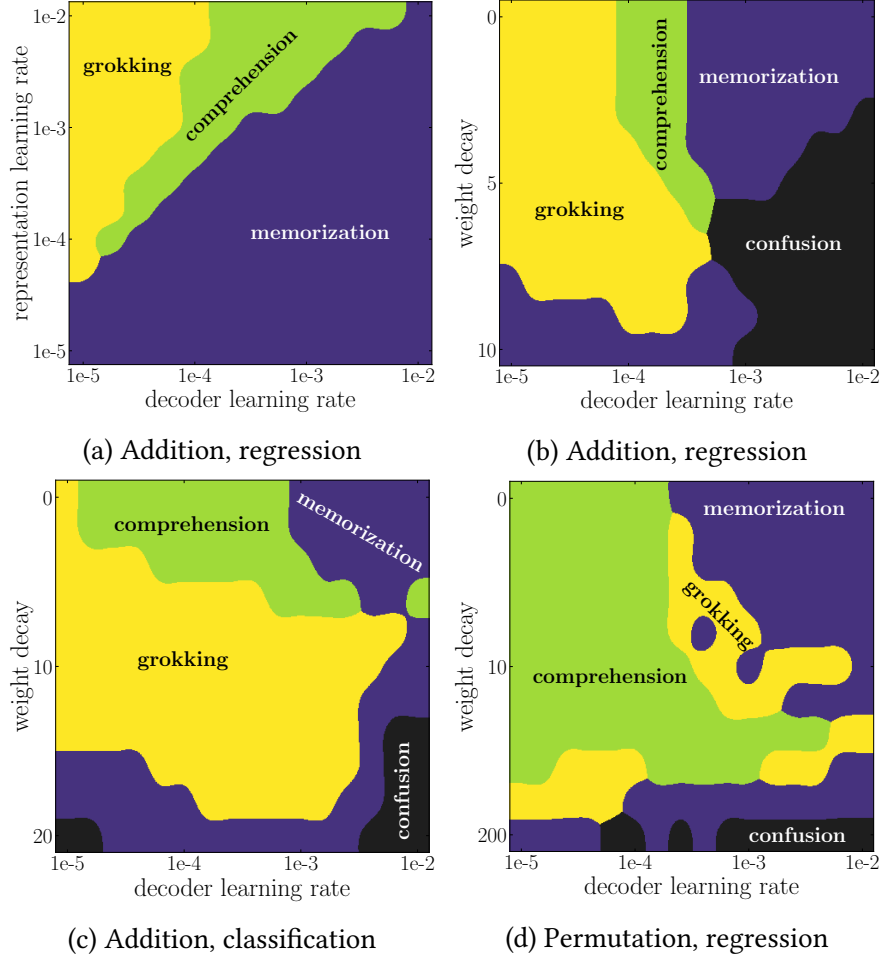


Figure 5.5: Phase diagrams of learning for the addition group and the permutation group. (a) shows the competition between representation and decoder. (b)(c)(d): each phase diagram contains four phases: comprehension, grokking, memorization and confusion, defined in Table 5.1. Grokking is sandwiched between comprehension and memorization.

Memorization is also called overfitting, and confusion means failure to even memorize training data. Figure 5.5 shows the phase diagrams for the addition group and the permutation group. They display quite rich phenomenon.

Competition between representation learning and decoder overfitting In the regression setup of the addition dataset, we show how the competition between representation learning rate and decoder learning rate lead to different learning phases in Figure 5.5 (a). As expected, a fast decoder coupled with slow representation learning (bottom right) lead to memorization. In the opposite extreme, although an extremely slow decoder coupled with fast representation learning (top left) will generalize in the end, the generalization time is long due to the inefficient decoder training. The ideal phase (comprehension) requires representation learning to be faster, but not too much, than the decoder.

Drawing from an analogy to physical systems, one can think of embedding vectors as a group of particles. In our effective theory from Section 5.3.2, internal particle interactions promote

Table 5.1: Definitions of the four phases of learning

Phase	criteria		
	training acc > 90% within 10^5 steps	validation acc > 90% within 10^5 steps	step(validation acc > 90%) -step(training acc > 90%) < 10^3
Comprehension	Yes	Yes	Yes
Grokking	Yes	Yes	No
Memorization	Yes	No	Not Applicable
Confusion	No	No	Not Applicable

the formation of structure, while external forces inhibit it. The decoder plays the role of an environment exerting external forces on the embeddings. If the decoder changes slowly/fast, the magnitude of the external forces are small/large, leading to better/worse representations.

Universality of phase diagrams We fix representation learning to be 10^{-3} and sweep instead decoder weight decay in Figure 5.5 (b)(c)(d). The phase diagrams correspond to addition regression (b), addition classification (c) and permutation regression (d), respectively. Common phenomenon emerge from these different tasks: (i) they all include four phases; (ii) The top right corner (a fast and capable decoder) is the memorization phase; (iii) the bottom right corner (a fast and simple decoder) is the confusion phase; (iv) grokking is sandwiched between comprehension and memorization, which seems to imply that it is an undesirable phase that stems from improperly tuned hyperparameters.

Due to limited space, we include phase diagrams of other hyperparameters (batch size, initialization scale and weight decay on representation) in Appendix 5.7.5.

5.4.2 Beyond the toy model

We conjecture that many of the principles which we saw dictate the training dynamics in the toy model also apply more generally. Below, we will see how our framework generalizes to transformer architectures for the task of addition modulo p , a minimal reproducible example of the original grokking paper [1].

We first encode $p = 53$ integers into 256D learnable embeddings, then pass two integers to a decoder-only transformer architecture. For simplicity, we do not encode the operation symbols here. The outputs from the last layer are concatenated and passed to a linear layer for classification. Training both the encoder and the decoder with the same optimizer (i.e., with the same hyperparameters) leads to the grokking phenomenon. Generalization appears much earlier once we lower the effective decoder capacity with weight decay (full phase diagram in Figure 5.6).

Early on, the model is able to perfectly fit the training set while having no generalization. We study the embeddings at different training times and find that neither PCA (shown in Figure 5.1) nor t-SNE (not shown here) reveal any structure. Eventually, validation accuracy starts to increase, and perfect generalization coincides with the PCA projecting the embeddings into a circle in 2D. Of course, no choice of dimensionality reduction is guaranteed to find any structure, and thus, it is challenging to show explicitly that generalization only occurs when a structure exists. Nevertheless, the fact that, when coupled with the implicit regularization of the optimizer for

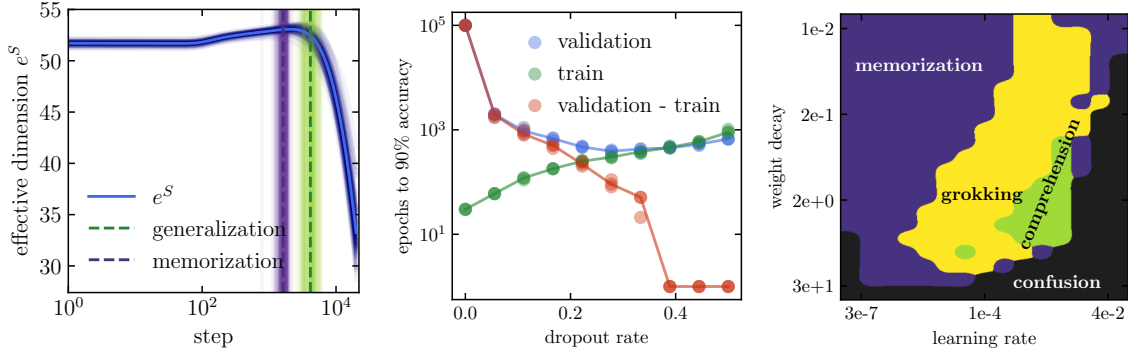


Figure 5.6: Left: Evolution of the effective dimension of the embeddings (defined as the exponential of the entropy) during training and evaluated over 100 seeds. Center: Effect of dropout on speeding up generalization. Right: Phase diagram of the transformer architecture. A scan is performed over the weight decay and learning rate of the decoder while the learning rate of the embeddings is kept fixed at 10^{-3} (with zero weight decay).

sparse solutions, such a clear structure appears in a simple PCA so quickly at generalization time suggests that our analysis in the toy setting is applicable here as well. This is also seen in the evolution of the entropy of the explained variance ratio in the PCA of the embeddings (defined as $S = -\sum_i \sigma_i \log \sigma_i$ where σ_i is the fractional variance explained by the i th principal component). As seen in Figure 5.6, the entropy increases up to generalization time then decreases drastically afterwards which would be consistent with the conjecture that generalization occurs when a low-dimensional structure is discovered. The decoder then primarily relies on the information in this low-dimensional manifold and essentially “prunes” the rest of the high-dimensional embedding space.

In Figure 5.6 (right), we show a comparable phase diagram to Figure 5.5 evaluated now in the transformer setting. Note that, as opposed to the setting in [1], weight decay has only been applied to the decoder and not to the embedding layer. Contrary to the toy model, a certain amount of weight decay proves beneficial to generalization and speeds it up significantly. We conjecture that this difference comes from the different embedding dimensions. With a highly over-parameterized setting, a non-zero weight decay gives a crucial incentive to reduce complexity in the decoder and help generalize in fewer steps. This is subject to further investigation. We also explore the effect of dropout layers in the decoder blocks of the transformer. With a significant dropout rate, the generalization time can be brought down to under 10^3 steps and the grokking phenomenon vanishes completely. The overall trend suggests that constraining the decoder with the same tools used to avoid overfitting reduces generalization time and can avoid the grokking phenomenon.

5.5 Related work

We are not aware of other formal attempts to understand grokking, though some authors have provided speculative, informal accounts. For instance, [2] describes training as a random walk over parameters, and draws connections to Solomonoff induction. Separately, [3] suggest that simpler functions which generalize achieve lower loss than complicated functions which memorize,

providing a training signal encouraging generalization. More broadly, our work is related to the following research directions:

Learning mathematical structures [4] trains a neural network to learn arithmetic operation from pictures of digits, but they do not observe grokking due to their abundant training data. Beyond arithmetic relations, machine learning has been applied to learn other mathematical structures, including geometry [5], knot theory [6] and group theory [7].

Double descent Grokking is somewhat reminiscent of the phenomena of “epoch-wise” *double descent* [8], where generalization can improve after a period of overfitting. [9] find that regularization can mitigate double descent, similar perhaps to how weight decay influences grokking.

Representation learning Representation learning lies at the core of machine learning [10–13]. Representation quality is usually measured by (perhaps vague) semantic meanings or performance on downstream tasks. In our study, the simplicity of arithmetic datasets allows us to define representation quality and study evolution of representations in a quantitative way.

Physics of learning Physics-inspired tools have been proved useful to understand machine learning from a theoretical perspective. These tools include effective theories [14,15], conservation laws [16] and free energy principle [17]. However, none of these works aim to develop an effective theory of representation learning.

We have argued that grokking occurs when models learn a particularly structured representation of their inputs. This connects a low-level understanding of the internal operation of our models with a high-level phenomenon: an increase in validation accuracy. In a recent work, researchers at Anthropic [18], connect a sudden decrease in loss during training with the emergence of *induction heads* within their models. They analogize their work to *statistical physics*, since it bridges a “microscopic”, mechanistic understanding of networks with “macroscopic” facts about overall model performance. We view our work in the same vein, but we make the additional contribution that we develop, in a toy setting, an **analytic model of low-level representation learning dynamics**.

5.6 Conclusion

We have shown how, in both toy models and more general settings, that representation enables generalization when it reflects structure in the data. We developed an effective theory of representation learning dynamics (in a toy setting) which predicts the critical dependence of learning on the training data fraction. We then presented phase transitions between comprehension, grokking, memorization and confusion where the learning “phase” depends on the decoder capacity and learning rate in decoder-only architectures. While we have mostly focused on a toy model, we find preliminary evidence that our results generalize to the setting of [1].

Our work can be viewed as a step towards a *statistical physics of deep learning*, a theory which connects the “microphysics” of low-level network dynamics with the “thermodynamics” of high-level model behavior. We view the application of theoretical tools from physics, such as effective theories [19], to be a rich area for further work. The broader impact of such work, if successful, would be to make models more transparent and predictable [18,20,21], crucial to the task of ensuring the safety of advanced AI systems.

Acknowledgements

We thank Liping Liu for helpful discussions. This work was supported by The Casey and Family Foundation, the Foundational Questions Institute, the Rothberg Family Fund for Cognitive Science and the NSF AI Institute for Artificial Intelligence and Fundamental Interactions (IAIFI) through NSF Grant No. PHY-2019786.

References

- [1] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra. “Grokking: Generalization beyond overfitting on small algorithmic datasets.” *arXiv preprint arXiv:2201.02177*, 2022.
- [2] B. Millidge. *Grokking ‘grokking’*. <https://beren.io/2022-01-11-Grokking-Grokking/>. 2022.
- [3] R. Shah. *Alignment Newsletter #159*. 2021. URL: https://www.alignmentforum.org/posts/zvWqPmQasssaAWkrj/an-159-building-agents-that-know-how-to-experiment-by%5C#DEEP_LEARNING_.
- [4] Y. Hoshen and S. Peleg. “Visual Learning of Arithmetic Operation.” In: *AAAI*. 2016.
- [5] Y.-H. He. “Machine-learning mathematical structures.” *arXiv preprint arXiv:2101.06317*, 2021.
- [6] S. Gukov, J. Halverson, F. Ruehle, and P. Sułkowski. “Learning to unknot.” *Machine Learning: Science and Technology*, 2(2), 2021, p. 025035.
- [7] A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, et al. “Advancing mathematics by guiding human intuition with AI.” *Nature*, 600(7887), 2021, pp. 70–74.
- [8] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever. “Deep double descent: Where bigger models and more data hurt.” *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12), 2021, p. 124003.
- [9] P. Nakkiran, P. Venkat, S. Kakade, and T. Ma. “Optimal regularization can mitigate double descent.” *arXiv preprint arXiv:2003.01897*, 2020.
- [10] Y. Bengio, A. Courville, and P. Vincent. “Representation learning: A review and new perspectives.” *IEEE transactions on pattern analysis and machine intelligence*, 35(8), 2013, pp. 1798–1828.
- [11] Y. Ouali, C. Hudelot, and M. Tami. “An overview of deep semi-supervised learning.” *arXiv preprint arXiv:2006.05278*, 2020.
- [12] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. Richemond, E. Buchatskaya, C. Doersch, B. Avila Pires, Z. Guo, M. Gheshlaghi Azar, et al. “Bootstrap your own latent-a new approach to self-supervised learning.” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 21271–21284.
- [13] P. H. Le-Khac, G. Healy, and A. F. Smeaton. “Contrastive representation learning: A framework and review.” *IEEE Access*, 8, 2020, pp. 193907–193934.
- [14] J. Halverson, A. Maiti, and K. Stoner. “Neural networks and quantum field theory.” *Machine Learning: Science and Technology*, 2(3), 2021, p. 035002.

- [15] D. A. Roberts, S. Yaida, and B. Hanin. “The principles of deep learning theory.” *arXiv preprint arXiv:2106.10165*, 2021.
- [16] D. Kunin, J. Sagastuy-Brena, S. Ganguli, D. L. Yamins, and H. Tanaka. “Neural mechanics: Symmetry and broken conservation laws in deep learning dynamics.” *arXiv preprint arXiv:2012.04728*, 2020.
- [17] Y. Gao and P. Chaudhari. “A free-energy principle for representation learning.” In: *International Conference on Machine Learning*. PMLR, 2020, pp. 3367–3376.
- [18] C. Olsson et al. “In-context Learning and Induction Heads.” *Transformer Circuits Thread*, 2022. <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [19] D. A. Roberts, S. Yaida, and B. Hanin. *The Principles of Deep Learning Theory*. <https://deeplearningtheory.com>. Cambridge University Press, 2022. arXiv: 2106.10165 [cs.LG].
- [20] D. Ganguli, D. Hernandez, L. Lovitt, N. DasSarma, T. Henighan, A. Jones, N. Joseph, J. Kernion, B. Mann, A. Askeel, et al. “Predictability and Surprise in Large Generative Models.” *arXiv preprint arXiv:2202.07785*, 2022.
- [21] J. Steinhardt. *Future ML Systems Will Be Qualitatively Different*. <https://www.lesswrong.com/s/4aARF2ZoBpFZAhhbe/p/pZaPhGg2hmmPwByHc>. 2022.
- [22] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. “Deep sets.” *Advances in neural information processing systems*, 30, 2017.

5.7 Appendix

5.7.1 Applicability of our toy setting

In the main paper, we focused on the toy setting with (1) the addition dataset and (2) the addition operation hard coded in the decoder. Although both simplifications appear to have quite limited applicability, we argue below that the analysis of the toy setting can actually apply to all Abelian groups.

The addition dataset is the building block of all Abelian groups A cyclic group is a group that is generated by a single element. A finite cyclic group with order n is $C_n = \{e, g, g^2, \dots, g^{n-1}\}$ where e is the identify element and g is the generator and $g^i = g^j$ whenever $i = j \pmod{n}$. The modulo addition and $\{0, 1, \dots, n-1\}$ form a cyclic group with $e = 0$ and g can be any number q coprime to n such that $(q, n) = 1$. Since algorithmic datasets contain only symbolic but no arithmetic information, the datasets of modulo addition could apply to all other cyclic groups, e.g., modulo multiplication and discrete rotation groups in 2D.

Although not all Abelian groups are cyclic, a finite Abelian group G can be always decomposed into a direct product of k cyclic groups $G = C_{n_1} \times C_{n_2} \cdots C_{n_k}$. So after training k neural networks with each handling one cyclic group separately, it is easy to construct a larger neural network that handles the whole Abelian group.

The addition operation is valid for all Abelian groups It is proved in [22] that for a permutation invariant function $f(x_1, x_2, \dots, x_n)$, there exists ρ and ϕ such that

$$f(x_1, x_2, \dots, x_n) = \rho\left[\sum_{i=1}^n \phi(x_i)\right], \quad (5.10)$$

or $f(x_1, x_2) = \rho(\phi(x_1) + \phi(x_2))$ for $n = 2$. Notice that $\phi(x_i)$ corresponds to the embedding vector $\mathbf{E}_i$, ρ corresponds to the decoder. The addition operator naturally emerges from the commutativity of the operator, not restricting the operator itself to be addition. For example, multiplication of two numbers x_1 and x_2 can be written as $x_1 x_2 = \exp(\ln(x_1) + \ln(x_2))$ where $\rho(x) = \exp(x)$ and $\phi(x) = \ln(x)$.

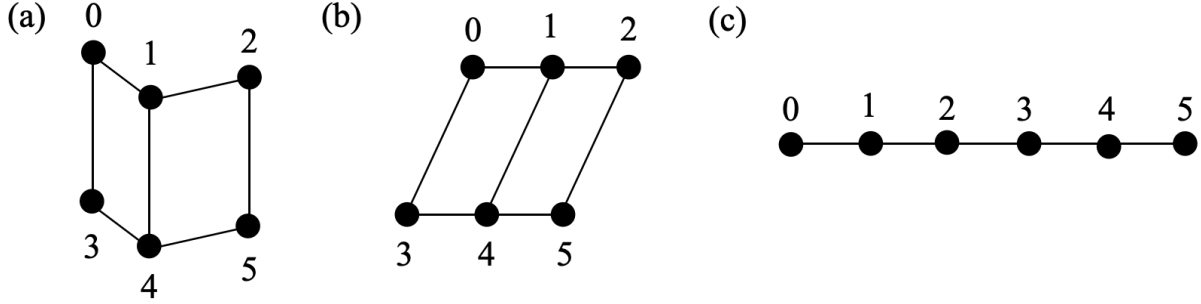


Figure 5.7: As we include more data in the training set, the (ideal) model have a more structured representation, from (a) to (b) to (c).

5.7.2 Definition of $\widehat{\text{Acc}}$

Given a training set D and a representation $\mathbf{R}$, if (i, j) is a validation sample, can the neural network correctly predict its output, i.e., $\text{Dec}(\mathbf{E}_i + \mathbf{E}_j) = \mathbf{Y}_{i+j}$? Since neural network has never seen (i, j) in the training set, one possible mechanism of induction is through

$$\text{Dec}(\mathbf{E}_i + \mathbf{E}_j) = \text{Dec}(\mathbf{E}_m + \mathbf{E}_n) = \mathbf{Y}_{m+n} (= \mathbf{Y}_{i+j}). \quad (5.11)$$

The first equality $\text{Dec}(\mathbf{E}_i + \mathbf{E}_j) = \text{Dec}(\mathbf{E}_m + \mathbf{E}_n)$ holds only when $\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n$ (i.e., (i, j, m, n) is a parallelogram). The second equality $\text{Dec}(\mathbf{E}_m + \mathbf{E}_n) = \mathbf{Y}_{m+n}$, holds when (m, n) in training set, i.e., $(m, n) \in D$, under the zero training loss assumption. Rigorously, given a training set D and a parallelogram set P (which can be calculated from $\mathbf{R}$), we collect all zero loss samples in an *augmented* training set $\overline{D}$

$$\overline{D}(D, P) = D \bigcup \{(i, j) | \exists (m, n) \in D, (i, j, m, n) \in P\}. \quad (5.12)$$

Keeping D fixed, a larger P would probably produce a larger $\overline{D}$, i.e., if $P_1 \subseteq P_2$, then $\overline{D}(D, P_1) \subseteq \overline{D}(D, P_2)$, which is why in Eq. (5.3) our defined $\text{RQI} \propto |P|$ gets its name “representation quality index”, because higher RQI normally means better generalization. Finally, the expected accuracy from a dataset D and a parallelogram set P is:

$$\widehat{\text{Acc}} = \frac{|\overline{D}(D, P)|}{|D_0|}, \quad (5.13)$$

which is the estimated accuracy (of the full dataset), and $P = P(\mathbf{R})$ is defined on the representation after training. On the other hand, accuracy Acc can be accessed empirically from trained neural

network. We verified $\text{Acc} \approx \widehat{\text{Acc}}$ in a toy setup (addition dataset $p = 10$, 1D embedding space, hard code addition), as shown in FIG. 5.3 (c). FIG. 5.3 (a)(b) show Acc and $\widehat{\text{Acc}}$ as a function of training set ratio, with each dot corresponding to a different random seed. The dashed red diagonal corresponds to memorization of the training set, and the vertical gap refers to generalization.

Although the agreement is good for 1D embedding space, we do not expect such agreement can trivially extend to high dimensional embedding space. In high dimensions, our definition of RQI is too restrictive. For example, suppose we have an embedding space with N dimensions. Although the representation may form a linear structure in the first dimension, the representation can be arbitrary in other $N - 1$ dimensions, leading to $\text{RQI} \approx 0$. However, the model may still generalize well if the decoder learns to keep only the useful information (the 1st dimension) and drop all other useless information (other 31 dimensions). It would be interesting to investigate how to define an RQI that takes into account the role of decoder in future works.

5.7.3 The gap of a realistic model $\mathcal{M}$ and the ideal model $\mathcal{M}^*$

Realistic models $\mathcal{M}$ usually form fewer number of parallelograms than ideal models $\mathcal{M}^*$. In this section, we analyze the properties of ideal models and calculated ideal RQI and ideal accuracy, which set upper bounds for empirical RQI and accuracy. The upper bound relations are verified via numerical experiments in Fig. 5.8.

Similar to Eq. (5.12) where some validation samples can be derived from training samples, we demonstrate how *implicit parallelograms* can be ‘derived’ from explicit ones in $P_0(D)$. The so-called derivation follows a simple geometric argument that: if A_1B_1 is equal and parallel to A_2B_2 , and A_2B_2 is equal and parallel to A_3B_3 , then we can deduce that A_1B_1 is equal and parallel to A_3B_3 (hence (A_1, B_2, A_2, B_1) is a parallelogram).

Recall that a parallelogram (i, j, m, n) is equivalent to $\mathbf{E}_i + \mathbf{E}_j = \mathbf{E}_m + \mathbf{E}_n$ (*). So we are equivalently asking if equation (*) can be expressed as a linear combination of equations in $A(P_0(D))$. If yes, then (*) is dependent on $A(P_0(D))$ (defined in Eq. (5.7)), i.e., $A(P_0(D))$ and $A(P_0(D) \cup (i, j, m, n))$ should have the same rank. We augment $P_0(D)$ by adding implicit parallelograms, and denote the augmented parallelogram set as

$$P(D) = P_0(D) \bigcup \{q \equiv (i, j, m, n) | q \in P_0, \text{rank}(A(P_0(D))) = \text{rank}(A(P_0(D) \bigcup q))\}. \quad (5.14)$$

We need to emphasize that an assumption behind Eq. (5.14) is that we have an ideal model $\mathcal{M}^*$. When the model is not ideal, e.g., when the injectivity of the encoder breaks down, fewer parallelograms are expected to form, i.e.,

$$P(R) \subseteq P(D). \quad (5.15)$$

The inequality is saying, whenever a parallelogram is formed in the representation after training, the reason is hidden in the training set. This is not a strict argument, but rather a belief that today’s neural networks can only copy what datasets (explicitly or implicitly) tell it to do, without any autonomous creativity or intelligence. For simplicity we call this belief *Alexander Principle*. In very rare cases when something lucky happens (e.g., neural networks are initialized at approximate correct weights), Alexander principle may be violated. Alexander principle sets an upper bound for RQI:

$$\text{RQI}(R) \leq \frac{|P(D)|}{|P_0|} \equiv \overline{\text{RQI}}, \quad (5.16)$$

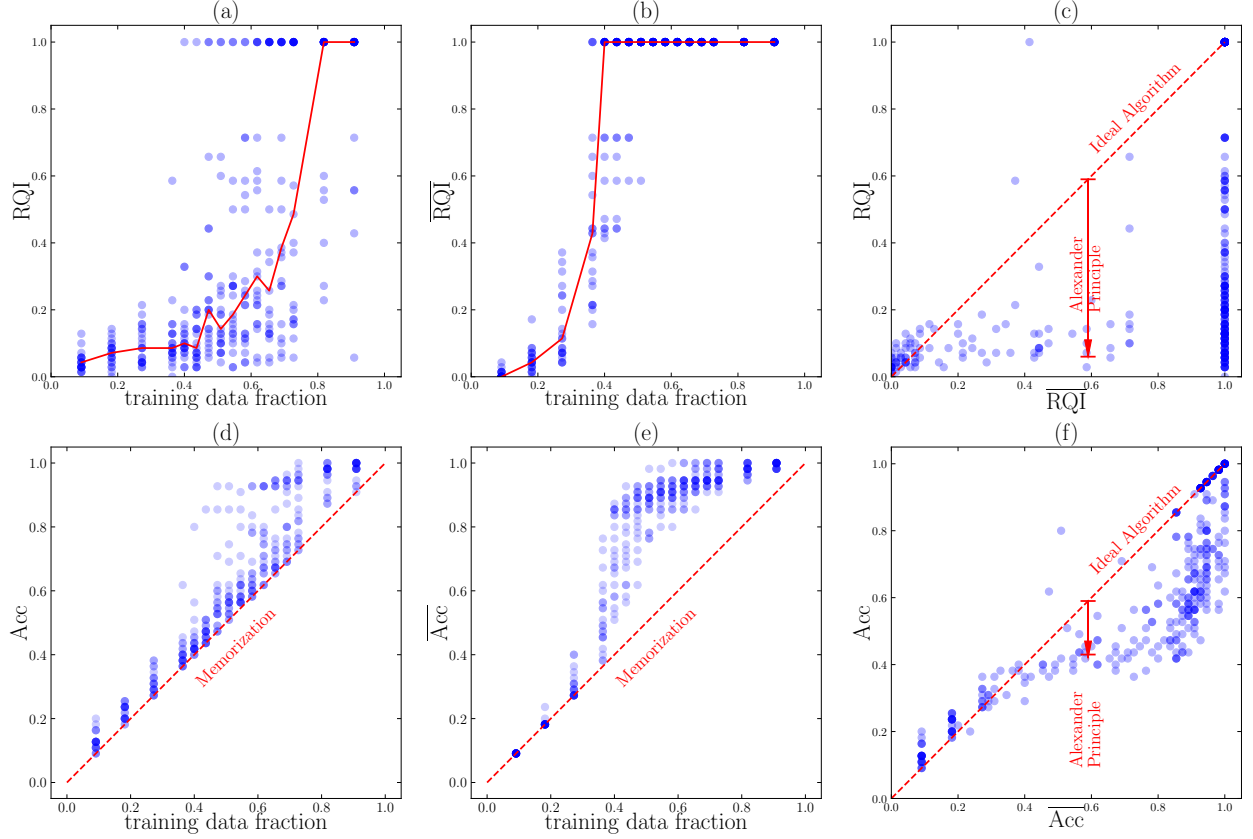


Figure 5.8: We compare RQI and Acc for an ideal algorithm (with bar) and a realistic algorithm (without bar). In (a)(b)(d)(e), four quantities (RQI , $\overline{\text{RQI}}$, Acc , $\overline{\text{Acc}}$) as functions of training data fraction are shown. In (c)(f), RQI and Acc of the ideal algorithm sets upper bounds for those of the realistic algorithm.

and sets an upper bound for $\widehat{\text{Acc}}$:

$$\widehat{\text{Acc}} \equiv \widehat{\text{Acc}}(D, P(R)) \leq \widehat{\text{Acc}}(D, P(D)) = \overline{\text{Acc}}, \quad (5.17)$$

In Figure 5.8 (c)(f), we verify Eq. (5.16) and Eq. (5.17). We choose $\delta = 0.01$ to compute $\text{RQI}(R, \delta)$. We find the trained models are usually far from being ideal, although we already include a few useful tricks proposed in Section 5.4 to enhance representation learning. It would be an interesting future direction to develop better algorithms so that the gap due to Alexander principle can be reduced or even closed. In Figure 5.8 (a)(b)(d)(e), four quantities (RQI , $\overline{\text{RQI}}$, Acc , $\overline{\text{Acc}}$) as functions of the training data fraction are shown, each dot corresponding to one random seed. It is interesting to note that it is possible to have $\overline{\text{RQI}} = 1$ only with $< 40\%$ training data, i.e., $55 \times 0.4 = 22$ samples, agreeing with our observation in Section 5.3.

Realistic representations Suppose an ideal model $\mathcal{M}^*$ and a realistic model $\mathcal{M}$ which train on the training set D give the representation R^* and R , respectively. What is the relationship between R and R^* ? Due to the Alexander principle we know $P(R) \subseteq P(D) = P(R^*)$. This means R^* has more parallelograms than R , hence R^* has fewer degrees of freedom than R .

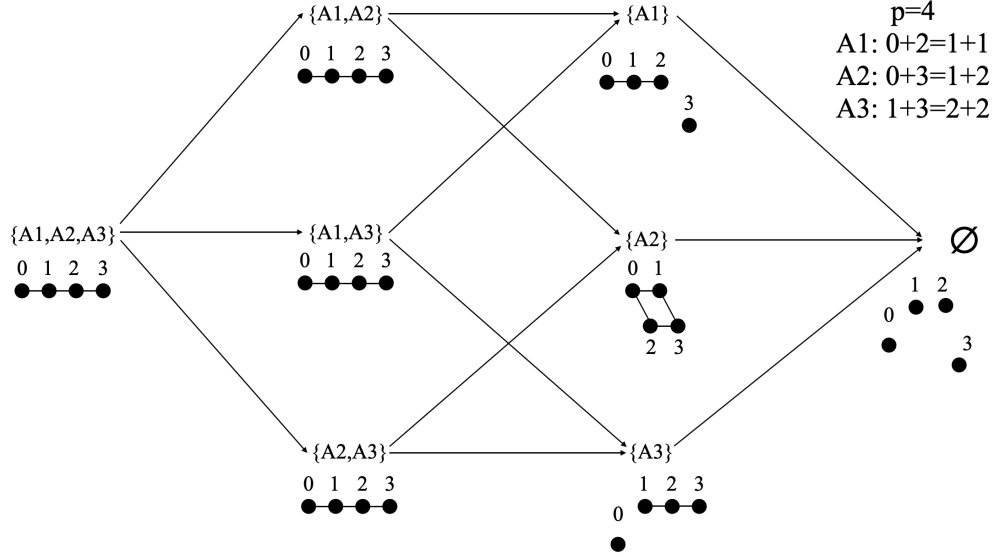


Figure 5.9: $p = 4$ case. Equation set A (or geometrically, representation) has a hierarchy: $a \rightarrow b$ means a is a parent of b , and b is a child of a . A realistic model can only generate representations that are descendants of the representation generated by an ideal model.

We illustrate with the toy case $p = 4$. The whole dataset contains $p(p+1)/2 = 10$ samples, i.e.,

$$D_0 = \{(0, 0), (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)\}. \quad (5.18)$$

The parallelogram set contains only three elements, i.e.,

$$P_0 = \{(0, 1, 1, 2), (0, 1, 2, 3), (1, 2, 2, 3)\}, \quad (5.19)$$

Or equivalently the equation set

$$A_0 = \{A1 : \mathbf{E}_0 + \mathbf{E}_2 = 2\mathbf{E}_1, A2 : \mathbf{E}_0 + \mathbf{E}_3 = \mathbf{E}_1 + \mathbf{E}_2, A3 : \mathbf{E}_1 + \mathbf{E}_3 = 2\mathbf{E}_2\}. \quad (5.20)$$

Pictorially, we can split all possible subsets $\{A | A \subseteq A_0\}$ into different levels, each level defined by $|A|$ (the number of elements). A subset A_1 in the i^{th} level points an direct arrow to another subset A_2 in the $(i+1)^{\text{th}}$ level if $A_2 \subset A_1$, and we say A_2 is a child of A_1 , and A_1 is a parent of A_2 . Each subset A can determine a representation R with $n(A)$ degrees of freedom. So R should be a descendant of R_* , and $n(R_*) \leq n(R)$. Numerically, $n(A)$ is equal to the dimension of the null space of A .

Suppose we have a training set

$$D = \{(0, 2), (1, 1), (0, 3), (1, 2), (1, 3), (2, 2)\}, \quad (5.21)$$

and correspondingly $P(D) = P_0$, $A(P) = A_0$. So an ideal model $\mathcal{M}_*$ will have the linear structure $\mathbf{E}_k = \mathbf{a} + k\mathbf{b}$ (see Figure 5.9 leftmost). However, a realistic model $\mathcal{M}$ may produce any descendants of the linear structure, depending on various hyperparameters and even random seeds.

In Figure 5.10, we show our algorithms actually generates all possible representations. We have two settings: (1) fast decoder $(\eta_1, \eta_2) = (10^{-3}, 10^{-2})$ (Figure 5.10 left), and (2) relatively slow decoder $(\eta_1, \eta_2) = (10^{-2}, 10^{-3})$ (Figure 5.10 right). The relatively slow decoder produces better representations (in the sense of higher RQI) than a fast decoder, agreeing with our observation in Section 5.4.

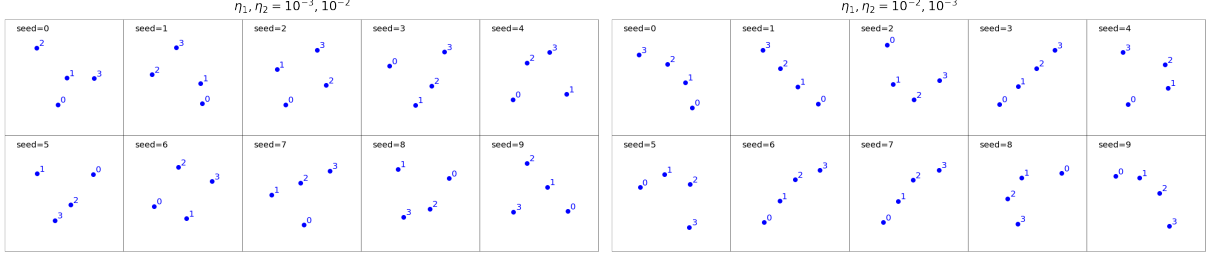


Figure 5.10: $p = 4$ case. Representations obtained from training neural networks are displayed. η_1 and η_2 are learning rates of the representation and the decoder, respectively. As described in the main text, $(\eta_1, \eta_2) = (10^{-2}, 10^{-3})$ (right) is more ideal than $(\eta_1, \eta_2) = (10^{-3}, 10^{-2})$ (left), thus producing representations containing more parallelograms.

5.7.4 Conservation laws of the effective theory

Recall that the effective loss function

$$\ell_{\text{eff}} = \frac{\ell_0}{Z_0}, \quad \ell_0 \equiv \sum_{(i,j,m,n) \in P_0(D)} |\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n|^2, \quad Z_0 \equiv \sum_k |\mathbf{E}_k|^2 \quad (5.22)$$

where ℓ_0 and Z_0 are both quadratic functions of $R = \{\mathbf{E}_0, \dots, \mathbf{E}_{p-1}\}$, and $\ell_{\text{eff}} = 0$ remains zero under rescaling and translation $\mathbf{E}'_i = a\mathbf{E}_i + \mathbf{b}$. The representation vector $\mathbf{E}_i$ evolves according to the gradient descent

$$\frac{d\mathbf{E}_i}{dt} = -\frac{\partial \ell_{\text{eff}}}{\partial \mathbf{E}_i}. \quad (5.23)$$

We will prove the following two quantities are conserved:

$$\mathbf{C} = \sum_k \mathbf{E}_k, \quad Z_0 = \sum_k |\mathbf{E}_k|^2 \quad (5.24)$$

Eq. (5.22) and Eq. (5.23) give

$$\frac{d\mathbf{E}_i}{dt} = -\frac{\ell_{\text{eff}}}{\partial \mathbf{E}_i} = -\frac{\partial(\frac{\ell_0}{Z_0})}{\partial \mathbf{E}_i} = -\frac{1}{Z_0} \frac{\partial \ell_0}{\partial \mathbf{E}_i} + \frac{\ell_0}{Z_0^2} \frac{\partial Z_0}{\partial \mathbf{E}_i}. \quad (5.25)$$

Then

$$\begin{aligned} \frac{dZ_0}{dt} &= 2 \sum_i \mathbf{E}_i \cdot \frac{d\mathbf{E}_i}{dt} \\ &= \frac{2}{Z_0^2} \sum_i \mathbf{E}_i \cdot (-Z_0 \frac{\partial \ell_0}{\partial \mathbf{E}_i} + \ell_0 \mathbf{E}_i) \\ &= \frac{2}{Z_0} (-\sum_k \frac{\partial \ell_0}{\partial \mathbf{E}_k} \cdot \mathbf{E}_k + \ell_0) \\ &= 0. \end{aligned} \quad (5.26)$$

where the last equation uses the fact that

$$\begin{aligned}
\sum_k \frac{\partial \ell_0}{\partial \mathbf{E}_k} \cdot \mathbf{E}_k &= 2 \sum_k \sum_{(i,j,m,n) \in P_0(D)} (\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n) (\delta_{ik} + \delta_{jk} - \delta_{mk} - \delta_{nk}) \cdot \mathbf{E}_k \\
&= 2 \sum_{(i,j,m,n) \in P_0(D)} (\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n) \sum_k (\delta_{ik} + \delta_{jk} - \delta_{mk} - \delta_{nk}) \cdot \mathbf{E}_k \\
&= \sum_{(i,j,m,n) \in P_0(D)} (\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n) \cdot (\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n) \\
&= 2\ell_0
\end{aligned}$$

The conservation of Z_0 prohibits the representation from collapsing to zero. Now that we have demonstrated that Z_0 is a conserved quantity, we can also show

$$\begin{aligned}
\frac{d\mathbf{C}}{dt} &= \sum_k \frac{d\mathbf{E}_k}{dt} \\
&= -\frac{1}{Z_0} \sum_k \frac{\partial \ell_0}{\partial \mathbf{E}_k} \\
&= -\frac{2}{Z_0} \sum_k \sum_{(i,j,m,n) \in P_0(D)} (\mathbf{E}_i + \mathbf{E}_j - \mathbf{E}_m - \mathbf{E}_n) (\delta_{ik} + \delta_{jk} - \delta_{mk} - \delta_{nk}) \\
&= \mathbf{0}.
\end{aligned} \tag{5.27}$$

The last equality holds because the two summations can be swapped and $\sum_k (\delta_{ik} + \delta_{jk} - \delta_{mk} - \delta_{nk}) = 0$.

5.7.5 More phase diagrams of the toy setup

We study another three hyperparameters in the toy setup by showing phase diagrams similar to Figure 5.5. The toy setup is: (1) addition without modulo ($p = 10$); (2) training/validation is split into 45/10; (3) hard code addition; (4) 1D embedding. In the following experiments, the decoder is an MLP with size 1-200-200-30. The representation and the encoder are optimized with AdamW with different hyperparameters. The learning rate of the representation is 10^{-3} . We sweep the learning rate of the decoder in range $[10^{-4}, 10^{-2}]$ as the x axis, and sweep another hyperparameter as the y axis. By default, we use full batch size 45, initialization scale $s = 1$ and zero weight decay of representation.

Batch size controls the amount of noise in the training dynamics. In Figure 5.11, the grokking region appears at the top left of the phase diagram (small decoder learning rate and small batch size). However, large batch size (with small learning rate) leads to comprehension, implying that smaller batch size seems harmful. This makes sense since to get crystals (good structures) in experiments, one needs a freezer which gradually decreases temperature, rather than something perturbing the system with noise.

Initialization scale controls distances among embedding vectors at initialization. We initialize components of embedding vectors from independent uniform distribution $U[-s/2, s/2]$ where s is called the initialization scale. Shown in Figure 5.12, it is beneficial to use a smaller initialization scale. This agrees with the physical intuition that closer particles are more likely to interact

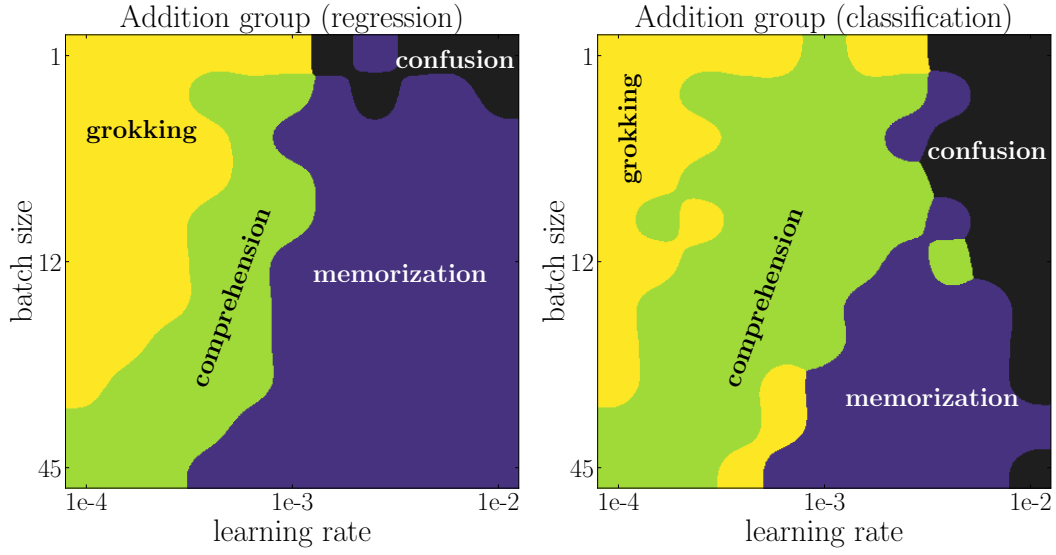


Figure 5.11: Phase diagrams of decoder learning rate (x axis) and batch size (y axis) for the addition group (left: regression; right: classification). Small decoder learning rate and large batch size (bottom left) lead to comprehension.

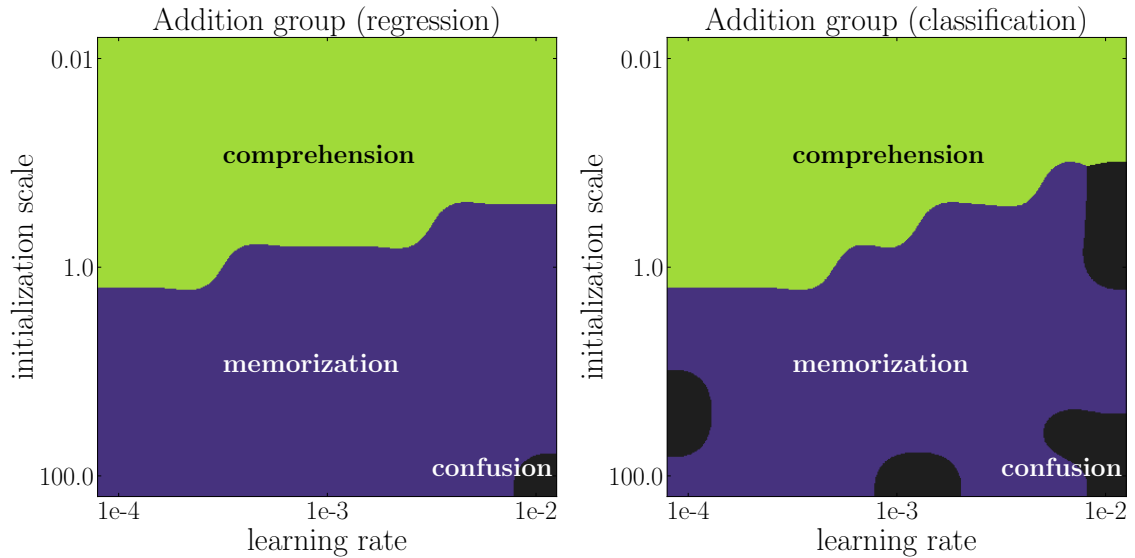


Figure 5.12: Phase diagrams of decoder learning rate (x axis) and initialization (y axis) for the addition group (left: regression; right: classification). Small initialization scale (top) leads to comprehension.

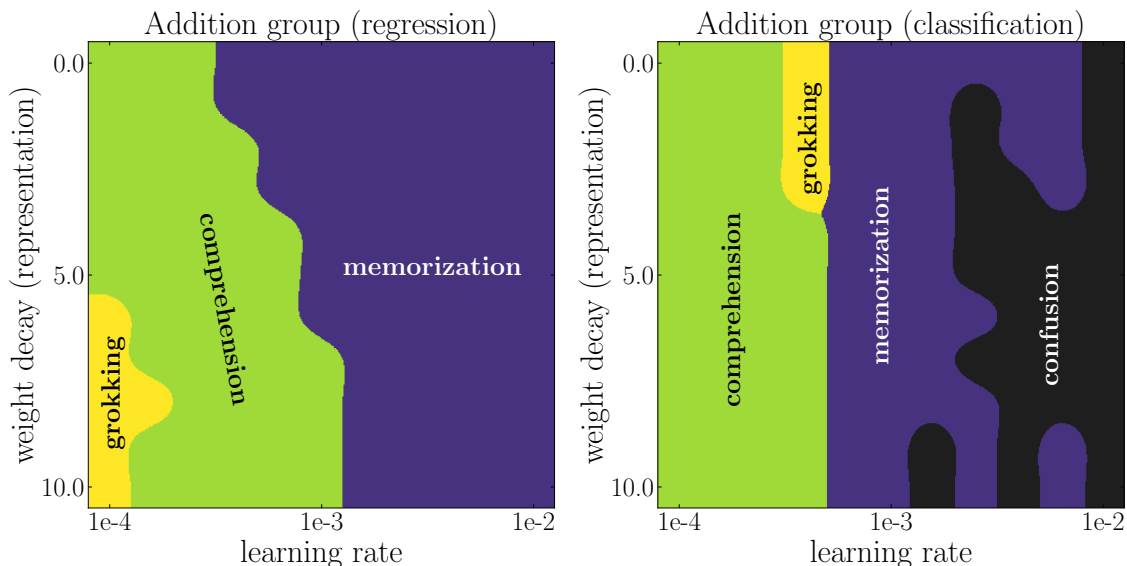


Figure 5.13: Phase diagrams of decoder learning rate (x axis) and representation weight decay (y axis) for the addition group (left: regression; right: classification). Representation weight decay does not affect model performance much.

and form structures. For example, the distances among molecules in ice are much smaller than distances in gas.

Representation weight decay controls the magnitude of embedding vectors. Shown in Figure 5.13, we see the representation weight decay in general does not affect model performance much.

5.7.6 General groups

Theory

We focused on Abelian groups for the most part of the paper. This is, however, simply due to pedagogical reasons. In this section, we show that it is straight-forward to extend definitions of parallelograms and representation quality index (RQI) to general non-Abelian groups. We will also show that most (if not all) qualitative results for the addition group also apply to the permutation group.

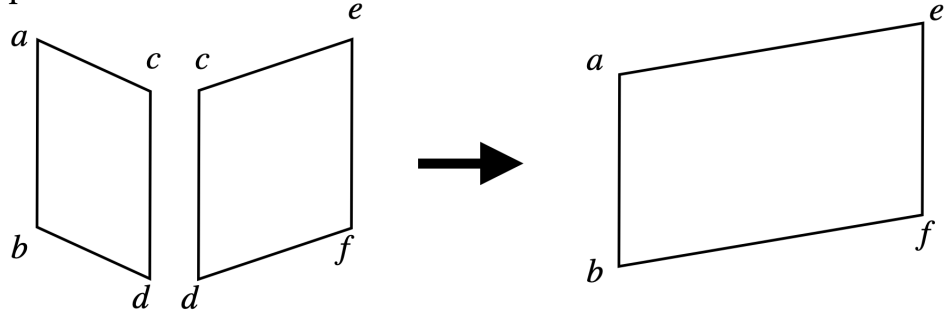
Matrix representation for general groups Let us first review the definition of group representation. A representation of a group G on a vector space V is a group homomorphism from G to $GL(V)$, the general linear group on V . That is, a representation is a map $\rho : G \rightarrow GL(V)$ such that

$$\rho(g_1 g_2) = \rho(g_1) \rho(g_2), \quad \forall g_1, g_2 \in G. \quad (5.28)$$

In the case V is of finite dimension n , it is common to identify $GL(V)$ with n by n invertible matrices. The punchline is that: each group element can be represented as a matrix, and the binary operation is represented as matrix multiplication.

A new architecture for general groups Inspired by the matrix representation, we embed each group element a as a learnable matrix $\mathbf{E}_a \in \mathbb{R}^{d \times d}$ (as opposed to a vector), and manually do

(a) Abelian Group: $2P \rightarrow P$



(b) Non-Abelian Group: $3P \rightarrow P$

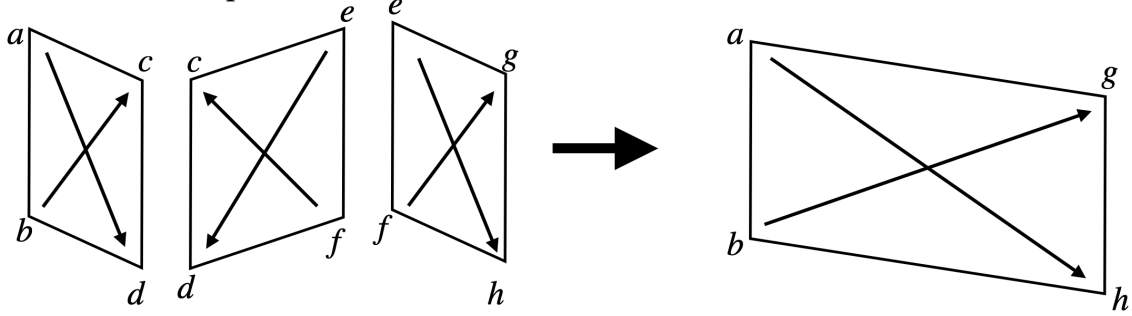


Figure 5.14: Deduction of parallelograms

matrix multiplication before sending the product to the decoder for regression or classification. More concretely, for $a \circ b = c$, our architecture takes as input two embedding matrices $\mathbf{E}_a$ and $\mathbf{E}_b$ and aims to predict $\mathbf{Y}_c$ such that $\mathbf{Y}_c = \text{Dec}(\mathbf{E}_a \mathbf{E}_b)$, where $\mathbf{E}_a \mathbf{E}_b$ means the matrix multiplication of $\mathbf{E}_a$ and $\mathbf{E}_b$. The goal of this simplification is to disentangle learning the representation and learning the arithmetic operation (i.e, the matrix multiplication). We will show that, even with this simplification, we are still able to reproduce the characteristic grokking behavior and other rich phenomenon.

Generalized parallelograms we define generalized parallelograms: (a, b, c, d) is a generalized parallelogram in the representation if $\|\mathbf{E}_a \mathbf{E}_b - \mathbf{E}_c \mathbf{E}_d\|_F^2 \leq \delta$, where $\delta > 0$ is a threshold to tolerate numerical errors. Before presenting the numerical results for the permutation group, we show an intuitive picture about how new parallelograms can be deduced from old ones for general groups, which is the key to generalization.

Deduction of parallelograms

We first recall the case of the Abelian group (e.g., addition group). As shown in Figure 5.14, when (a, d, b, c) and (c, f, d, e) are two parallelograms, we have

$$\begin{aligned} \mathbf{E}_a + \mathbf{E}_d &= \mathbf{E}_b + \mathbf{E}_c \\ \mathbf{E}_c + \mathbf{E}_f &= \mathbf{E}_d + \mathbf{E}_e \end{aligned} \tag{5.29}$$

We can derive that $\mathbf{E}_a + \mathbf{E}_f = \mathbf{E}_b + \mathbf{E}_e$ implying that (a, f, b, e) is also a parallelogram. That is, for Abelian groups, two parallelograms are needed to deduce a new parallelogram.

For the non-Abelian group, if we have only two parallelograms such that

$$\begin{aligned} \mathbf{E}_a \mathbf{E}_d &= \mathbf{E}_b \mathbf{E}_c \\ \mathbf{E}_f \mathbf{E}_c &= \mathbf{E}_e \mathbf{E}_d \end{aligned} \quad (5.30)$$

we have $\mathbf{E}_b^{-1} \mathbf{E}_a = \mathbf{E}_c \mathbf{E}_d^{-1} = \mathbf{E}_f^{-1} \mathbf{E}_e$, but this does not lead to something like $\mathbf{E}_f \mathbf{E}_a = \mathbf{E}_e \mathbf{E}_b$, hence useless for generalization. However, if we have a third parallelogram such that

$$\mathbf{E}_e \mathbf{E}_h = \mathbf{E}_f \mathbf{E}_g \quad (5.31)$$

we have $\mathbf{E}_b^{-1} \mathbf{E}_a = \mathbf{E}_c \mathbf{E}_d^{-1} = \mathbf{E}_f^{-1} \mathbf{E}_e = \mathbf{E}_g \mathbf{E}_h^{-1}$, equivalent to $\mathbf{E}_a \mathbf{E}_h = \mathbf{E}_b \mathbf{E}_g$, thus establishing a new parallelogram (a, h, b, g) . That is, for non-Abelian groups, three parallelograms are needed to deduce a new parallelogram.

Numerical Results

In this section, we conduct numerical experiments on a simple non-abelian group: the permutation group S_3 . The group has 6 group elements, hence the full dataset contains 36 samples. We embed each group element a into a learnable 3×3 embedding matrix $\mathbf{E}_a$. We adopt the new architecture described in the above subsection: we hard code matrix multiplication of two input embedding matrices before feeding to the decoder. After defining the generalized parallelogram in the last subsection, we can continue to define RQI (as in Section 5.3) and predict accuracy $\widehat{\text{Acc}}$ from representation (as in appendix 5.7.2). We also compute the number of steps needed to reach $\text{RQI} = 0.95$.

Representation We flatten each embedding matrix into a vector, and apply principal component analysis (PCA) to the vectors. We show the first three principal components of these group elements in Figure 5.15. On the plane of PC1 and PC3, the six points are organized as a hexagon.

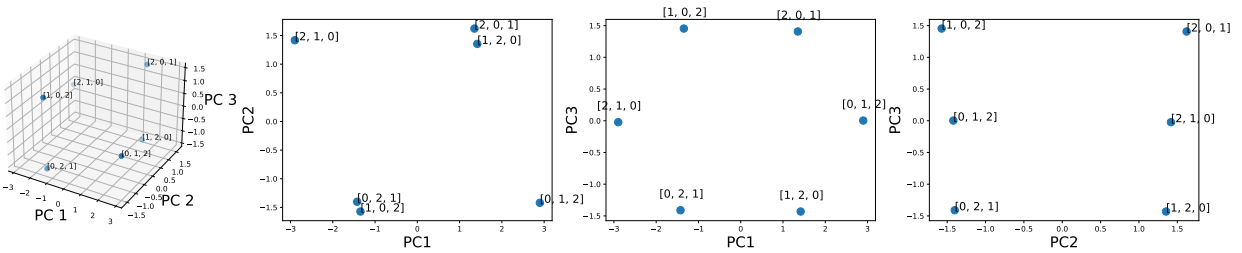


Figure 5.15: Permutation group S_3 . First three principal components of six embedding matrices $\mathbb{R}^{3 \times 3}$.

RQI In Figure 5.16(a), we show RQI as a function of training data fraction. For each training data fraction, we run 11 random seeds (shown as scatter points), and the blue line corresponds to the highest RQI.

Steps to reach RQI= 0.95 In Figure 5.16(b), we show the steps to reach $\text{RQI} > 0.95$ as a function of training data fraction, and find a phase transition at $r = r_c = 0.5$. The blue line corresponds to the best model (smallest number of steps).

Accuracy The real accuracy Acc is shown in Figure 5.16(c), while the predicted accuracy $\widehat{\text{Acc}}$ (calculated from RQI) is shown in Figure 5.16(d). Their comparison is shown in (e): $\widehat{\text{Acc}}$ is a lower bound of Acc , implying that there must be some generalization mechanism beyond RQI.

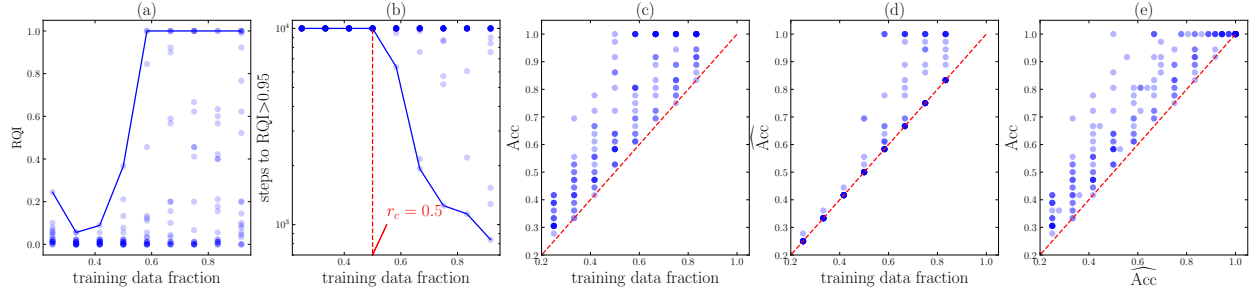


Figure 5.16: Permutation group S_3 . (a) RQI increases as training set becomes larger. Each scatter point is a random seed, and the blue line is the highest RQI obtained with a fixed training set ratio; (b) steps to reach RQI > 0.95. The blue line is the smallest number of steps required. There is a phase transition around $r_c = 0.5$. (c) real accuracy Acc; (d) predicted accuracy Acc; (e) comparison of Acc and Acc: Acc serves as a lower bound of Acc.

Phase diagram We investigate how the model performance varies under the change of two knobs: decoder learning rate and decoder weight decay. We calculate the number of steps to training accuracy ≥ 0.9 and validation accuracy ≥ 0.9 , respectively, shown in Figure 5.5 (d).

Chapter 6

Omnigrok: Grokking Beyond Algorithmic Data^{*}

In this paper, we develop another perspective on “grokking”. We observe that there is a relationship, in practice, between how well networks generalize and the size of their weight norm. We find that network parameters θ with high weight norm $||\theta||$ but which fit the training set well tend to generalize worse than parameters θ with lower weight norm. We find that we can induce grokking on datasets where it doesn’t normally occur by increasing the initial weight norm of networks, and we can accelerate generalization by constraining neural network weight norm to be small. See Section 6.8.5 for commentary on how this work on grokking relates to other theories from the literature and the core ideas of this thesis.

Abstract

Grokking, the unusual phenomenon for algorithmic datasets where generalization happens long after overfitting the training data, has remained elusive. We aim to understand grokking by analyzing the loss landscapes of neural networks, identifying the mismatch between training and test loss landscapes as the cause for grokking. We refer to this as the “LU mechanism” because training and test losses (against model weight norm) typically resemble “L” and “U”, respectively. This simple mechanism can nicely explain many aspects of grokking: data size dependence, weight decay dependence, the emergence of representations, etc. Guided by the intuitive picture, we are able to induce grokking on tasks involving images, language and molecules. In the reverse direction, we are able to eliminate grokking for algorithmic datasets. We attribute the dramatic nature of grokking for algorithmic datasets to representation learning.

^{*}Ziming Liu, Eric J. Michaud, and Max Tegmark. “Omnigrok: Grokking beyond algorithmic data.” *International Conference on Learning Representations* (2023).

6.1 Introduction

Generalization lies at the heart of machine learning. A good machine learning model should arguably be able to generalize fast, and behave in a smooth/predictable way under changes of (hyper)parameters. Grokking, the phenomenon where the model generalizes long after overfitting the training set, has raised interesting questions after it was observed on algorithmic datasets by [1]:

Q1 *The origin of grokking*: Why is generalization much delayed after overfitting?

Q2 *The prevalence of grokking*: Can grokking occur on datasets other than algorithmic datasets?

This paper aims to answer these questions by analyzing neural loss landscapes:

A1 Grokking is caused by the mismatch between training and test loss landscapes. Specifically, (reduced) training and test losses plotted against model weight norm resemble "L" and "U", respectively, as shown in Figure 6.1b. We refer to this phenomenon as the "LU mechanism", which we elaborate on in Section 6.2 and 6.3.

A2 **Yes**. Indeed, we demonstrate grokking for a wide range of machine learning tasks in Section 6.4, including image classification, sentiment analysis and molecule property prediction. Grokking signals observed for these tasks are usually less dramatic than for algorithmic datasets, which we attribute to representation learning in Section 6.5.

Partial answers to **Q1** are provided in recent studies: [2] attribute grokking to the slow formation of good representations, [3] attempts to link grokking to the slingshot mechanism of adaptive optimizers, and [4] uses Fourier gap to describe hidden progress. This paper aims to understand grokking through the lens of neural loss landscapes. Our landscape analysis is able to explain many aspects of grokking: data size dependence, weight decay dependence, emergence of representations, etc.

The paper is organized as follows: In Section 6.2, we review background on generalization, and introduce the *LU mechanism*. In Section 6.3, we show how the LU mechanism leads to grokking for a toy teacher-student setup. In Section 6.4, we show that the intuition gained from the toy problem can transfer to realistic datasets (MNIST, IMDB reviews and QM9), for which we also observe grokking, although in a slightly non-standard setup where it is relatively weak. In Section 6.5, we discuss why grokking is more dramatic for algorithmic datasets than on others (e.g., MNIST), by comparing their loss landscapes. As a byproduct, we find that training with constrained weight norm can almost eliminate grokking. We review related work in Section 6.6 and summarize our conclusions in Section 6.7. Code is available at <https://github.com/KindXiaoming/Omnigrok>.

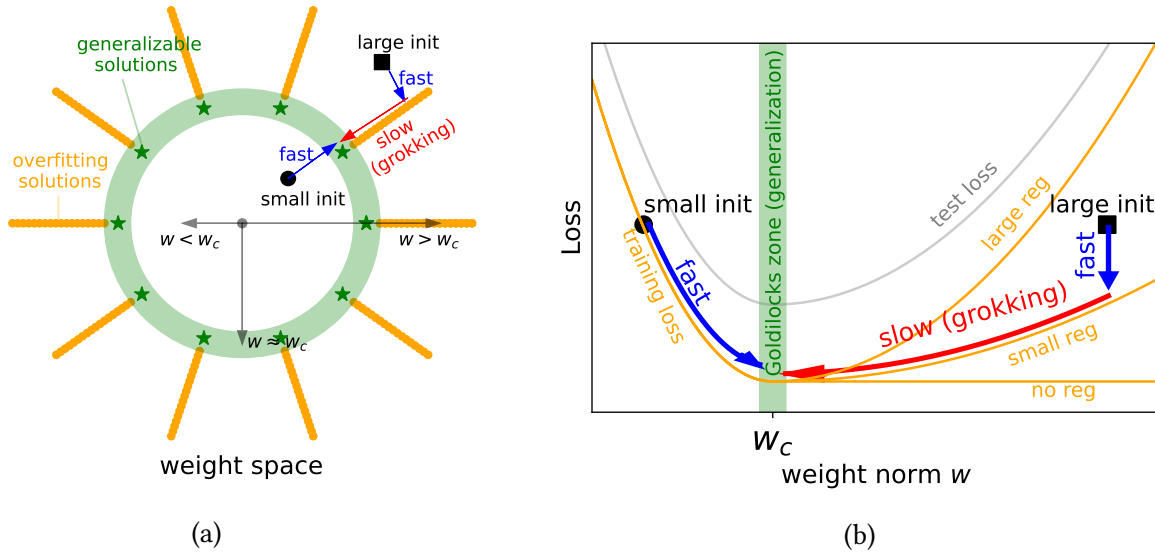


Figure 6.1: (a) w : L_2 norm of model weights. Generalizing solutions (green stars) are concentrated around a sphere in the weight space where $w \approx w_c$ (green). Overfitting solutions (orange) populate the $w \gtrsim w_c$ region. (b) The training loss (orange) and test loss (gray) have the shape of L and U, respectively. Their mismatch in the $w > w_c$ region leads to fast-slow dynamics, resulting in grokking.

6.2 The LU mechanism for grokking

Weight norm and reduced loss Letting $\mathbf{w}$ denote the weights of a model, any function $f(\mathbf{w})$ (e.g., train/test loss/accuracy) depends on both the weight norm $w \equiv \|\mathbf{w}\|_2$ and the angular direction $\hat{\mathbf{w}} \equiv \mathbf{w}/w$. Similar to [5], we define a reduced function $\tilde{f}(w)$ by minimizing training loss $l_{\text{train}}(\mathbf{w})$ over angular directions, i.e.,

$$\tilde{f}(w) \equiv f(\mathbf{w}^*(w)), \quad \text{where } \mathbf{w}^*(w) \equiv \underset{\|\mathbf{w}\|_2=w}{\operatorname{argmin}} l_{\text{train}}(\mathbf{w}) \quad (6.1)$$

In practice, we perform the constrained minimization by rescaling the model weights back to their original norm after each unconstrained optimization step. We will see that this reduced 1D loss landscape, which is easy to visualize, captures important features related to grokking. Throughout the paper, our model is initialized by multiplying a factor $\alpha \equiv w/w_0$ to the standard initialization¹, where w_0 and w are the weight norm of the network before and after multiplying α .

LU mechanism Although the loss landscapes of neural networks are nonlinear, [5] reveal a simple landscape picture: There is a spherical shell in the weight space (the "Goldilocks" zone), where generalization is better than outside this zone. We illustrate the Goldilocks zone as the green area with average radius w_c in Figure 6.1a; the green stars are the generalizing solutions. The test loss is thus higher either both when $w > w_c$ and $w < w_c$, forming a U-shape against w in Figure 6.1b (gray curve). By contrast, the training loss has an L-shape against weight norm². There are many solutions which overfit training data for $w > w_c$, but high training losses are

¹The standard initialization means the default one in PyTorch.

² $l_{\text{train}}(w)$ is non-increasing: One can always construct a larger network by adding useless non-zero weights to a smaller network, without changing the functionality of the model.

incurred for $w < w_c$. This corresponds to the L-shaped curve seen in Figure 6.1b (orange curve, no regularization). In summary, the (reduced) training loss and test loss are L-shaped and U-shaped against weight norm, respectively, which we will refer to as the LU mechanism throughout the paper.

It is well known in statistics that generalization error has a "U" shape against model capacity, which is usually attributed to the bias-variance trade-off. Although this common wisdom was challenged by the observation of double descent [6], the "U" curve can be recovered from a double descent simply by changing the x-axis from the number of model parameters N to the 2-norm of model parameters $w \equiv \|\mathbf{w}\|_2$ [7]. Although the LU mechanism may remind readers of related phenomena [6,8,9], their setups are not exactly the same as ours. More importantly, our focus and contribution is to understand grokking, a brand new generalization puzzle.

Grokking dynamics We identify the "LU mechanism" as the cause of grokking. If the weight norm is initialized to be large (e.g., the black square in the $w > w_c$ region), the model first quickly moves to a nearby overfitting solution by minimizing the training loss. Without any regularization, the model will stay where it is, because the gradient of the training loss is almost zero along the valley of overfitting solutions, so generalization does not happen. Fortunately, there are usually explicit and/or implicit regularizations that can drive the weight vector towards the Goldilocks zone $w \approx w_c$. When the regularization magnitude is non-zero but small, the radial motion can be (arbitrarily) slow. If weight decay is the only source of regularization, and training loss is negligible after overfitting, then weight decay γ causes $w(t) \approx \exp(-\gamma t)w_0$, when $w_0 > w_c$, so it takes time $t \approx \ln(w_0/w_c)/\gamma \propto \gamma^{-1}$ to generalize. A small γ results in a huge generalization delay (i.e., grokking). The dependence on regularization magnitudes is illustrated in Figure 6.1b: no generalization at all happens for $\gamma = 0$, small γ leads to slow generalization (grokking), and large γ leads to faster generalization³. The above analysis only applies to large initializations $w > w_c$. Small initializations $w < w_c$ can always generalize fast⁴, regardless of regularization.

Why isn't grokking commonly observed? The standard initialization schemes typically initialize w no larger than w_c . However, if we increase initialization scales (explicitly or implicitly), grokking can appear. In Section 6.3 and 6.4, we find that explicitly increasing initialization weight norm can induce grokking. In Section 6.5, we argue for algorithmic datasets because (shown in Figure 6.6d)

$$w_c(\text{bad representation}) > w_c(\text{good representation}), \quad (6.2)$$

i.e., a proper initialization for a bad representation is effectively too large for a good representation, leading to grokking. Take the addition (base p) for example: with the good (linear) representation or a bad (random) representation, the decoder needs to learn to classify $O(p)$ or $O(p^2)$ examples, respectively.

6.3 Grokking for a teacher-student setup

To illustrate how the LU mechanism results in grokking, we employ a toy teacher-student setup. The teacher and the student share the same architecture (a 5-100-100-5 MLP with tanh activation), but are initialized with different seeds. The student network is initialized with the standard

³ γ should not be too large, otherwise it will bring the weights to a trivial solution $\mathbf{w} = \mathbf{0}$.

⁴ w should not be too small to harm optimization.

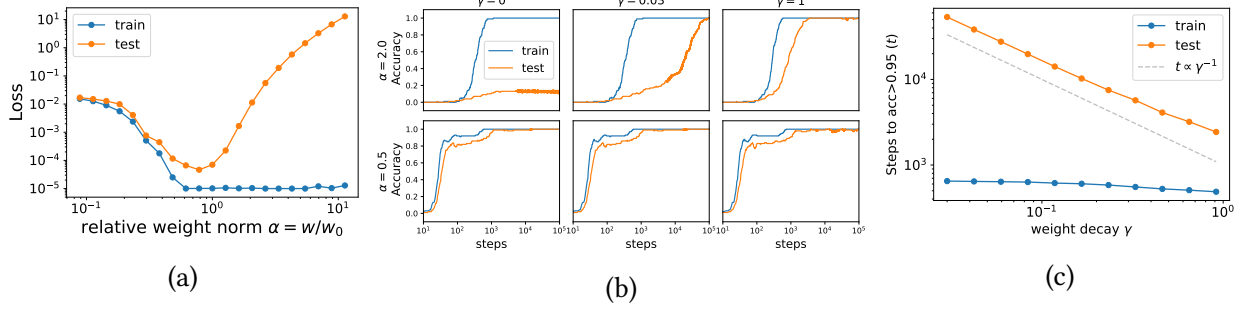


Figure 6.2: Teacher-student setup. α : student initialization scale, γ : weight decay. (a) The reduced training loss and test loss have the shape of "L" and "U", respectively. (b) Top row: large initialization ($\alpha = 2.0$) can demonstrate no generalization (no reg), grokking (small reg) and fast generalization (large reg). Bottom: small initialization ($\alpha = 0.5$) always generalizes fast, regardless of weight decay. (c) $\alpha = 2.0$. The steps to overfitting is independent of weight decay, while the steps to generalization scale inversely with the weight decay.

initialization (the default one in PyTorch) but each weight is rescaled by the same factor $\alpha \equiv w/w_0$, where w_0 and w are the weight norm of the student network before and after rescaling. The teacher network is initialized standardly, i.e., $\alpha_{\text{teacher}} = 1$. Inputs and outputs have dimensions $d_{\text{in}} = 5$ and $d_{\text{out}} = 5$, respectively. We generate $N_{\text{train}} = 100$ training and $N_{\text{test}} = 100$ test samples by first drawing inputs from the standard Gaussian distribution $N(0, \mathbf{I}_{d_{\text{in}} \times d_{\text{in}}})$, and then feed the input data to the teacher to generate output labels. The student network is trained with the Adam optimizer (learning rate 3×10^{-4}) for 10^5 steps.

LU landscapes Firstly, we compute the reduced losses by minimizing the training loss (excluding weight decay) while constraining the weight norm of the student network to be constant. We treat the converging point after training as the global minimum on the spherical surface, which explicitly defines the reduced losses $\tilde{l}_{\text{train}}(\alpha)$ and $\tilde{l}_{\text{test}}(\alpha)$. As shown in Figure 6.2a, $\tilde{l}_{\text{test}}(\alpha)$ first decreases and then increases as α increases, displaying a U-shape with a minimum at $\alpha \approx 1$. By contrast, $\tilde{l}_{\text{train}}(\alpha)$ decreases when $\alpha < 1$ and remains flat near zero when $\alpha \geq 1$, forming an L-shape. When weight decay γ is present, the training landscape becomes $\tilde{l}_{\text{train}}(\alpha, \gamma) = \tilde{l}_{\text{train}}(\alpha) + \gamma \alpha^2 C^2$ where C is the average parameter magnitude determined by the standard initialization.

Training dynamics Our problem is a regression task, but we can imitate the behavior of a classification task by manually setting a threshold $\theta = 0.01$ and defining a sample to be correctly "classified" if the prediction error is less than θ . We study the dynamics of training and test accuracy. We run experiments with two initializations $\alpha = 0.5$ (small) and $\alpha = 2.0$ (large), and three weight decays $\gamma = 0$ (no reg), $\gamma = 0.03$ (small reg) and $\gamma = 1$ (large reg). As shown in Figure 6.2b (bottom), small initialization runs always generalize fast regardless of regularization. Large initialization runs (top) depended on weight decay: no regularization fails to generalize, small regularization generalizes slowly (grokking), while large regularization generalizes faster.

For the large initialization $\alpha = 2.0$, we do a finer sweep of γ in $[0.03, 1]$. We compute the number of steps and weight norm w when training or test accuracy reaches 95%. As shown in Figure 6.2c, the time (number of steps) to reach 95% training accuracy is independent of weight decay γ , while the time to reach 95% test accuracy is inversely proportional to the weight decay, as we derived above for the LU mechanism.

6.4 Omnigrok: Grokking for more interesting tasks

We now analyze loss landscapes and search for grokking for several more interesting datasets, and see that the insights obtained from our toy model can transfer to these datasets. We report the main results here, with experiment details included in Appendix 6.8.1.

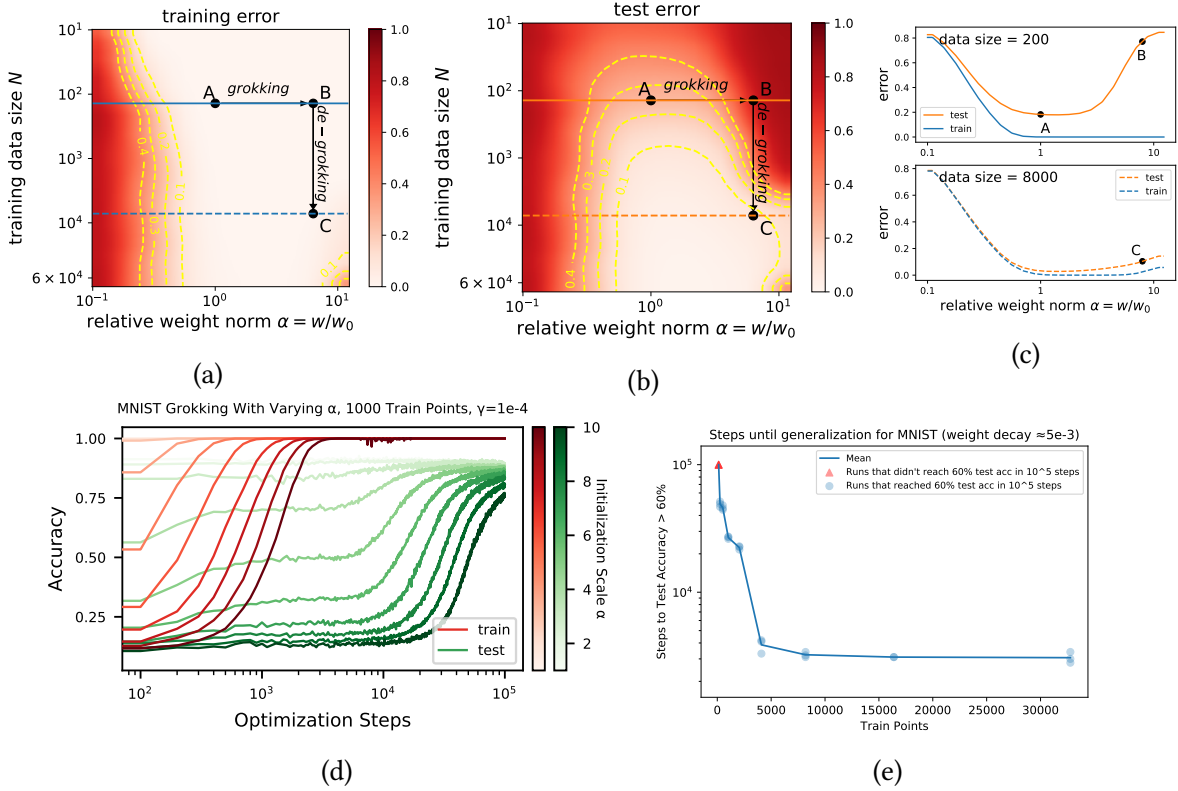


Figure 6.3: MNIST. (a) reduced training error, (b) reduced test error. Comparing A and B: larger weight norm makes learning grok (delay generalization). Comparing B and C: a larger training data size makes learning de-grok (speed up generalization). (c) "LU" holds truer for smaller data. (d) Accuracy curves for MNIST in the setting where we observe grokking. (e) Time to generalize as a function of training set size N .

Image classification We visualize loss landscapes of MNIST [10] to verify the LU mechanism, and study the dependence on training data size. Similar to the teacher-student case, we reduce losses and errors (one minus accuracy) to two variables (weight norm w and data size N) by minimizing over angular directions of weights, i.e.,

$$\tilde{l}_{\text{train}}(w, N) \equiv l_{\text{train}}(\mathbf{w}^*, N), \quad \tilde{l}_{\text{test}}(w, N) \equiv l_{\text{test}}(\mathbf{w}^*, N), \quad \mathbf{w}^*(w, N) \equiv \underset{\|\mathbf{w}\|_2 = w}{\operatorname{argmin}} l_{\text{train}}(\mathbf{w}, N), \quad (6.3)$$

shown in Figure 6.3 (a)(b). The reduced loss landscape reveals three things: (1) Larger initializations lead to grokking. Point A in Figure 6.3 corresponds to the standard initialization ($\alpha = 1$), which has low training and test errors, hence no grokking. When increasing the weight norm from A to B, training error is seen to remain low while test error rises. To generalize, implicit or

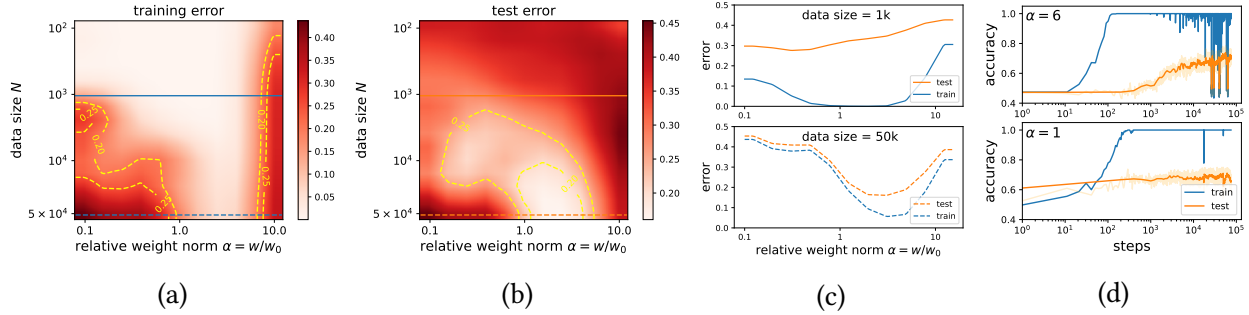


Figure 6.4: We use an LSTM to predict IMDB reviews. (a) training error; (b) test error; (c) reduced losses for data size 1k (top) and 50k (bottom); (d) With 1k data, a (weak) grokking signal is observed for large initializations ($\alpha = 6$), while no grokking is observed for standard initializations ($\alpha = 1$).

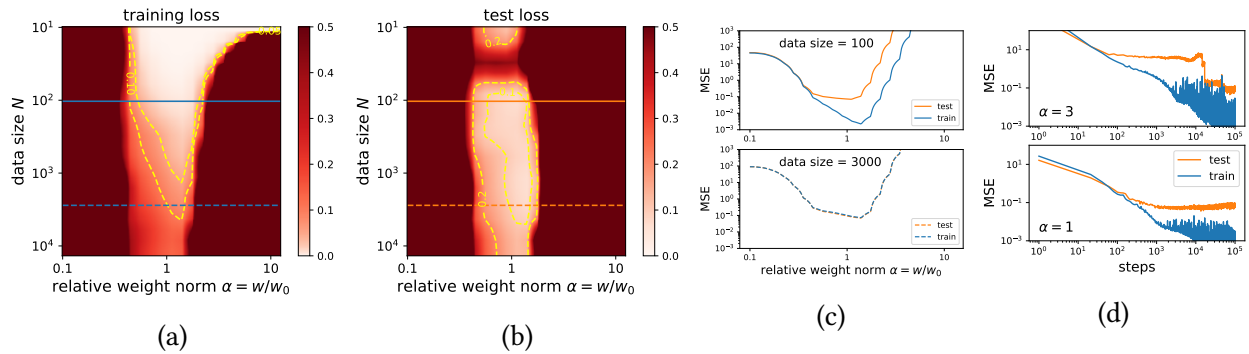


Figure 6.5: We use a GCNN to predict isotropic polarizability of molecules in the QM9 dataset. (a) training loss; (b) test loss; (c) reduced losses for data size 100 (top) and 3000 (bottom); (d) with 200 training samples, grokking is observed for large initialization ($\alpha = 3$), while no grokking is observed for standard initializations ($\alpha = 1$).

explicit regularization such as weight decay then brings the weight norm down, leading to delayed generalization (grokking) if regularization is small. (2) Larger datasets lead to de-grokking. Comparing B and C in Figure 6.3, C is seen to have larger training size than B and lower test error. Larger data size N makes the Goldilocks zone broader, reducing or eliminating grokking even for large weight initializations. (3) Critical data size can be defined. As reported in [1,2], we see that there exists a critical training set size below which generalization is impossible. The effective theory analysis in [2] only applies to algorithmic datasets, but not to other datasets with unknown optimal representations. The loss landscape analysis presented in this work can apply to all supervised-learning tasks. As shown in Figure 6.3 (b), the contours of constant test error are thumb-like, and the tip of the thumb determines the minimum amount of data required for generalization.

Guided by the landscape analysis, we make two nonstandard decisions to induce grokking on MNIST: (1) we reduce the size of the training set from 60k to 1k samples (by taking a random subset) and (2) we increase the scale of the weight initialization distribution (by multiplying the initial weights, sampled with Kaiming uniform initialization, by a constant $\alpha > 1$). With these modifications to the training set size and initialization scale, we train a depth-3 width-200 MLP

with ReLU activations with the AdamW optimizer using MSE loss with one-hot targets. We find that the network quickly fits the training set, and test accuracy improves much later, as shown in Figure 6.3d, just as in the stereotypical grokking learning first observed in algorithmic datasets. Figure 6.3e shows the effect of training set size on time to generalization for MNIST. We find a result similar to what [1] observed, namely that generalization time increases rapidly once one approaches a certain critical data set size.

Sentiment analysis of text We look for grokking using LSTMs [11] for IMDb dataset [12]. Similar to Eq. (6.3), we reduce training and test losses to depend on only the weight norm w and data size N . We show the reduced training and test error in Figure 6.4 (a)(b). For large data size say the full dataset, training and test errors have similar "U" shapes⁵, so one cannot create grokking via the "LU" mechanism. For small data size, say 1k, however, the mismatch between training and test errors makes it possible to create grokking via large initializations. In Figure 6.4 (c), we initialize weights larger ($\alpha = 6$) with weight decay 1, overfitting is complete within 10^2 steps, but generalization does not start until around 10^3 steps. Note that the generalization "jump" is not as sharp as on algorithmic datasets [1] or MNIST, but at least generalization is delayed here. By contrast, if we use the standard initialization ($\alpha = 1$) with no weight decay, generalization happens early on during training, and does not improve much after overfitting.

Molecules We search for grokking using the graph convolutional neural network (GCNN) for QM9 dataset [13]. Similar to Eq. (6.3), we define the reduced training/test losses, which are only dependent on weight norm w and data size N . As shown in Figure 6.5(a)(b), when data size is large, training and test losses have similar "U" shapes, hence grokking is impossible via the "LU mechanism". When data size is small, training and test losses mismatch somewhere in the region $\alpha = w/w_0 > 1$, making grokking possible. Indeed, shown in Figure 6.5(d), there is a sharp drop in test loss around 10^4 steps if initialization is 3 times larger than standard, while standard initialization does not lead to grokking. Note that zero weight decay is applied in both cases, implying the existence of implicit regularizations.

6.5 Representation is key to grokking

In Section 6.4, we showed that increasing initialization scales can make grokking happen for standard ML tasks. However, this seems a bit artificial and does not explain why standard initialization leads to grokking on algorithmic datasets, but not on standard ML datasets, say MNIST. The key difference is how much the task relies on representation learning. For the MNIST dataset, the quality of representation determines whether the test accuracy is 95% or 100%; by contrast in algorithmic datasets, the quality of representation determines whether test accuracy is random guess (bad representation) or 100% (good representation). So overfitting (under a bad representation) has a more dramatic effect on algorithmic datasets, i.e., the model weights increase quickly during overfitting but test accuracy remains low. During overfitting, model weight norm is much larger than at initialization, but then drops below the initialization norm when the model generalizes, shown in Figure 6.7a, and also observed by [14]. As a byproduct, we are able to eliminate grokking by constraining the model on a small weight norm sphere, shown in Figure 6.7b.

⁵In principle, reduced training losses should be non-increasing ("L"), but optimization issues may occur for too large initializations [8].

In the following, we will compare algorithmic datasets (Section 6.5.1) to MNIST (Section 6.5.2). We show how their loss landscapes depend on representations differently, and how the difference leads to different outcomes (grokking or not).

6.5.1 Algorithmic datasets

Setup We take the toy addition setup in [2], where each input digit $0 \leq i \leq p-1$ (output label $0 \leq k \leq 2(q-1)$) is embedded as a vector $\mathbf{E}_i$ ($\mathbf{Y}_k$). A decoder MLP is employed to predict $\mathbf{Y}_k = \text{Dec}(\mathbf{E}_i + \mathbf{E}_j)$ ($k = i + j$). In the setup of grokking, both the decoder and the input representations $\mathbf{R} \equiv \{\mathbf{E}_i\}$ are trainable, with learning rates η_D and η_R , respectively; in the setup of landscape analysis, only decoder is trainable, as we explain below. Training and test losses depend on three factors: (i) representation $\mathbf{R}$, (ii) weight norm w and (iii) weight direction $\hat{\mathbf{w}}$. As in previous sections, we can optimize $\hat{\mathbf{w}}$ by minimizing the training loss on constant weight norm spheres. We further reduce the high-dimensional representations to 1D by interpolating in a particular direction:

$$\mathbf{R} = m\mathbf{R}_{\text{random}} + (1 - m)\mathbf{R}_{\text{linear}} \quad (6.4)$$

where $\mathbf{R}_{\text{linear}}$ refers to the linear representation in which number k is embedded to $\mathbf{E}_k = [k, 0, \dots, 0]$, $\mathbf{R}_{\text{random}}$ is drawn from Gaussian distributions, i.e., $\mathbf{E}_k \sim N(\mathbf{0}, \mathbf{I})$, and $m \in [0, 1]$ is a scalar interpolating between $\mathbf{R}_{\text{linear}}$ and $\mathbf{R}_{\text{random}}$, that we term *representation messiness* because $\mathbf{R} = \mathbf{R}_{\text{linear}}$ when $m = 0$, and $\mathbf{R} = \mathbf{R}_{\text{random}}$ when $m = 1$. After these reductions, both training and test losses become functions of two variables, representation messiness m and weight norm w :

$$\mathbf{w}^*(w, m) \equiv \underset{\|\mathbf{w}\|_2=w}{\operatorname{argmin}} l_{\text{train}}(\mathbf{w}, m), \quad \tilde{l}_{\text{train}}(w, m) \equiv l_{\text{train}}(\mathbf{w}^*, m), \quad \tilde{l}_{\text{test}}(w, m) \equiv l_{\text{test}}(\mathbf{w}^*, m) \quad (6.5)$$

Note that our definition of $\tilde{l}_{\text{train}}(w, m)$ excludes the weight decay term $\ell_{\text{reg}} = \frac{1}{2}\gamma w^2$, but we should be aware of its presence when we analyze the dynamics of (w, m) , which is governed by the gradient flow on $\tilde{l}_{\text{train}}(w, m)$ plus weight decay (η_R/η_D are learning rates of representation/decoder):

$$\frac{dw}{dt} = -\eta_D \left(\frac{\partial \tilde{l}_{\text{train}}}{\partial w} + \gamma w \right), \quad \frac{dm}{dt} = -\eta_R \frac{\partial \tilde{l}_{\text{train}}}{\partial m}, \quad (6.6)$$

Landscape We show $\tilde{l}_{\text{train}}(w, m)$ and $\tilde{l}_{\text{test}}(w, m)$ in Figures 6.6a and 6.6b, indicating the generalizing solution with a green star. Based on the reduced training loss (Figure 6.6a), we can divide the 2D plane into two regions I and II, separated by a dashed yellow line (the contour of training loss = 0.05): (I): The darker region, with high training losses/gradients and small weight norm. (II): The lighter region, with low training losses/gradients and large weight norm. Comparing Figures 6.6a and 6.6b reveals that training and test loss landscapes differ, especially in region II. Moreover, while the training loss depends weakly on m , the test loss depends strongly on m . As we will see, the (weak) dependence of training loss on representation drives the model to the generalizing solution. However, the driving force is small because the dependence is weak, leading to grokking. We elaborate below how these particular loss landscapes lead to grokking.

Grokking dynamics In region II, the dynamics is slow (for small γ) due to nearly vanishing gradients. By contrast, the dynamics in region I is relatively fast. As we will explain, dynamics is

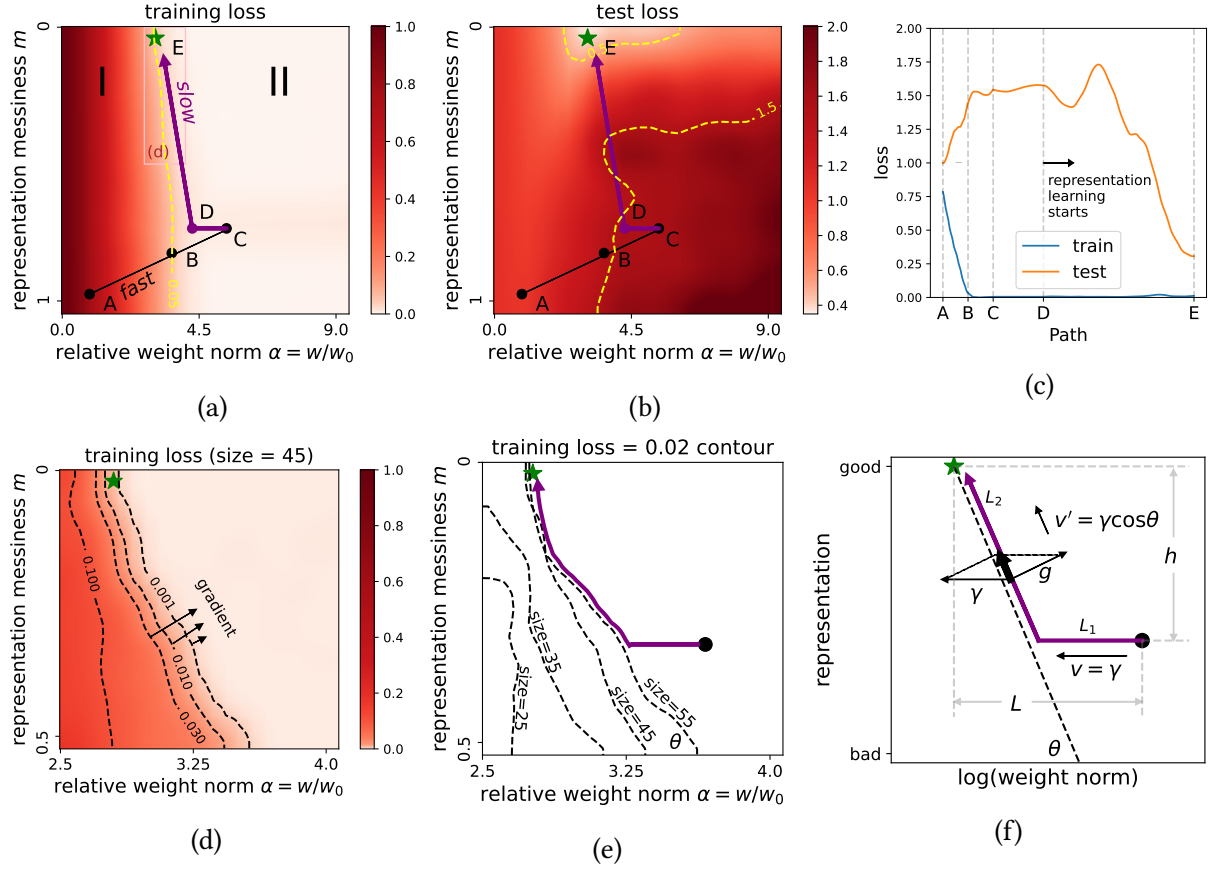


Figure 6.6: Loss landscapes for a toy MLP, on the 2D (w, m) plane. (a) Training loss splits the plane into two regions: large loss small w (fast dynamics) and small loss large w (slow dynamics). (b) Test loss; the green star is the generalizing solution. (c) Losses along an illustrative path A $\rightarrow$ E, demonstrating multiple descent; (d) zoom-in of the training loss highlighting the gradients on the boundary. (e) the boundary depends on training data size; (f) a simple illustration of grokking dynamics.

also slow on the boundary of I and II, and grokking is the consequence of traversing region II and/or the boundary.

Let us analyze a typical path A to E shown in Figure 6.6(a)(b). A rolls "downhill" to B following training gradients, possibly continuing to C due to momentum. C is located in II where $\tilde{l}_{\text{train}} \approx 0$, so according to Eq. (6.6), $dm/dt \approx 0$ and $dw/dt \approx -\eta_D \gamma w$ or, equivalently, $d(\log w)/dt \approx -\eta_D \gamma$. So $(\log w, m)$ moves with a constant speed $v = \eta_D \gamma$ in the $-w$ direction from C to D, a point near the boundary. Negative gradients around the boundary point towards larger w and smaller m , shown in Figure 6.6d (a zoom-in of Figure 6.6a). The gradients become increasingly large as the model goes deeper inside region I, and at some point, the gradient totally cancels out v in the gradient direction, making the model start to drift along the boundary, as illustrated in Figure 6.6f. Then the model moves along the boundary with a new velocity $v' = v \cos \theta$ ⁶, until it reaches the generalizing solution E.

⁶For simplicity, we assume $\eta_R = \eta_D$ here, but the analysis can apply to any (η_R, η_D) .

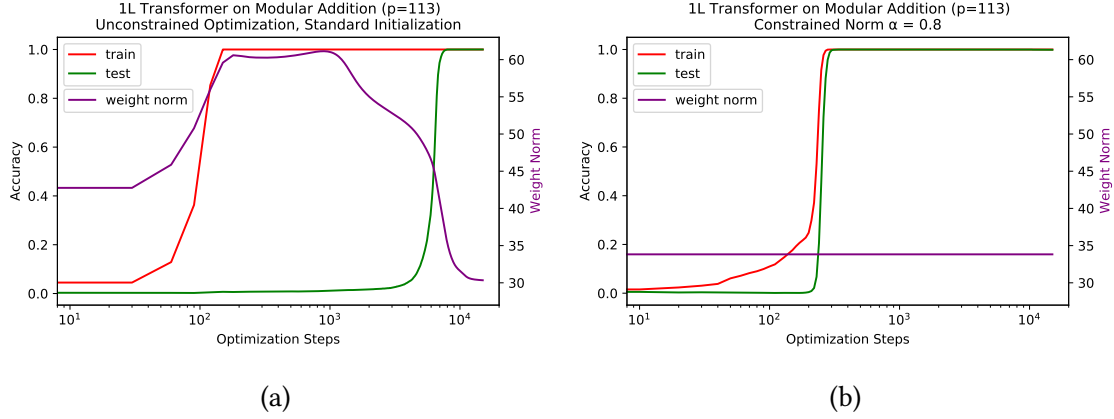


Figure 6.7: Training 1L transformer on modular addition ($p = 113$). (a) Weight norm, train accuracy, and test accuracy over time, initialized and trained normally. Weight norm first increases, and is highest during the period of overfitting, but then drops to become lower than initial weight norm when the model generalizes. (b) Constrained optimization at constant weight norm ($\alpha = 0.8$) largely eliminates grokking, with test and train accuracy improving concurrently.

The slow dynamics from C to E is the origin of grokking. During this period, the model first moves in the $-w$ direction with a velocity v over the distance $L_1 = L - h \cot \theta$, and then moves along the boundary with a velocity v' over the distance $L_2 = h / \sin \theta$. So the total time is $t = L_1 / v + L_2 / v' = (L + h \tan \theta) / (\eta_D \gamma)$. This formula agrees with the observation that large weight decays γ and/or larger decoder learning rates η_D can make generalization happen faster [1, 2]. Besides, the path manifests intriguing multiple descent of test loss, shown in Figure 6.6c.

The above picture is supported by a transformer experiment: Figure 6.7a, shows how model norm changes over time and we see that there is an initial increase in weight norm, which peaks during overfitting, but then drops during the period of generalization to be lower than the initialization norm. For this experiment, we used the setup of [14], training a 1-layer transformer on modular addition ($p = 113$). The model width $d_{\text{model}} = 128$, with 4 attention heads, and $d_{\text{mlp}} = 512$ with ReLU activations. We train with AdamW with a learning rate of 0.001 and weight decay $\gamma = 1$.

Dependence of grokking on training data size Another important observation in [1] is that grokking happens faster for larger training size. Our landscape analysis can also explain the data size dependence. In Figure 6.6e, we show the contours (training loss = 0.02) for different training sizes (25, 35, 45, 55). The contours of training size 45 and 55 both connect to the green star, meaning that generalization will eventually happen. However, the slopes of the contours are different, i.e., $\theta_{55} < \theta_{45}$. Since $t = (L + h \tan \theta) / (\eta_D \gamma)$ increases as θ increases, we have $t_{55} < t_{45}$, i.e., more training data leads to faster grokking. For training size 35 and 25, the contours do not connect to the green star, so generalization will not happen, no matter how long the training will be run.

De-grokking by constraining weight norm Guided by our understanding of grokking from the LU mechanism, we find that we can control grokking in the setting where it was first observed by [1] – transformers trained on algorithmic tasks. As shown in Figure 6.7b, reducing the initialization scale and constraining optimization to hold model weight norm constant over training brings

train accuracy and test accuracy learning curves together, almost eliminating grokking.

6.5.2 MNIST

We now study how training and test losses depend on representation messiness in the MNIST dataset. We denote the 28×28 images as the raw representation $\mathbf{R}_{\text{raw}}$. We construct a linearly separable representation $\mathbf{R}_{\text{linear}}$ by assigning input representations proportional to their label y_i , for example, an image of a 2 is represented by a matrix with all elements being 2. Similar to Eq. (6.4), we use $m \in [0, 1]$ to interpolate between $\mathbf{R}_{\text{raw}}$ and $\mathbf{R}_{\text{linear}}$:

$$\mathbf{R} = m\mathbf{R}_{\text{raw}} + (1 - m)\mathbf{R}_{\text{linear}}, \quad (6.7)$$

Similarly to Eq. (6.5), we define and plot $\tilde{l}_{\text{train}}(w, m)$ and $\tilde{l}_{\text{test}}(w, m)$ in Figure 6.8, using the full dataset $N = 60000$. Comparing Figures 6.8a and 6.8b reveals two things: (1) The training and test losses behave similarly; (2) Both training and test losses depend very weakly on m . This implies that the raw image representation is already quite close to being optimal, so decent test accuracy can be obtained even without learning optimal representations. As a result, grokking does not occur (Figure 6.8c).

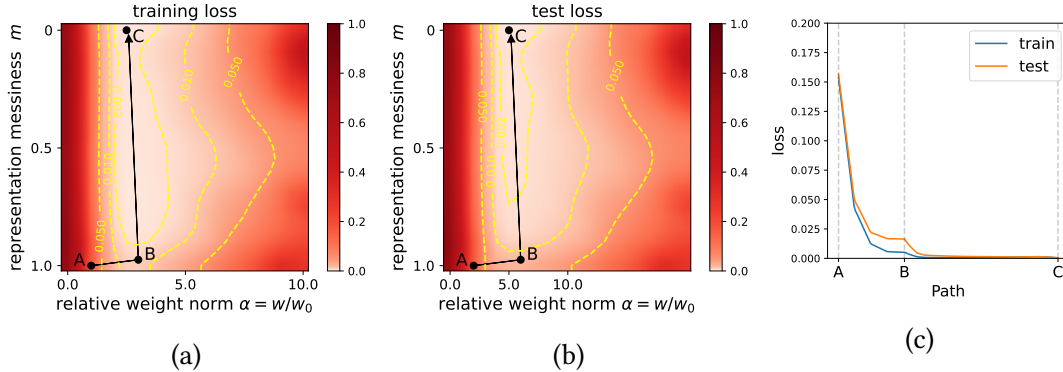


Figure 6.8: MNIST landscapes as functions of representation messiness m and weight norm w : (a) training loss, and (b) test loss. Training and test losses do not have significant mismatch, and neither of them depend on representation strongly, which is in stark contrast to algorithmic datasets (Figure 6.6). (c) an illustrative path $A \rightarrow B \rightarrow C$ does not manifest grokking.

Comparing Figure 6.6 and 6.8, we see that the (strong) dependence of test performance on the representation is the key to grokking: the dependence on representation is strong for algorithmic datasets, so grokking happens. By contrast, the dependence is weak for MNIST, so grokking does not happen.

Discussion: grokking on language models? We conjecture that grokking is more easily observed in tasks where generalization relies heavily on learning good representations (from scratch). This seems to imply the possibility of grokking on language tasks where word embeddings are key to generalization. However, we have not yet observed clear grokking signals for large language models, perhaps because: (i) the structure of languages is complicated, so the "optimal representation" for language might be much "messier" than algorithmic representations. (ii) Pre-training avoids learning representations from scratch, hence helps reduce possible grokking.

6.6 Relation to Related works

Grokking was first observed for algorithmic datasets by [1]. Several formal or informal attempts have been made to understand grokking: (a) [2] attributes grokking to the slow formation of good representations. (b) [15] suggests that generalizable solutions achieve lower loss than overfitting solutions, providing a training signal encouraging generalization. (c) [14] suggests grokking is a phase change due to limited data and regularization. (d) [4] suggests that generalization is due not to random search, but to hidden progress of SGD to gradually amplify a Fourier gap. (e) [3] links grokking to the "Slingshot mechanism" specific to adaptive optimizers. (f) [16] describes training as a random walk over parameters. Our conclusion supports (a)(b)(c)(d), but does not necessarily negate (e)(f).

Double descent is the phenomenon that performance first gets worse and then gets better as we increase the model size, data size, training epochs or regularization [6,17]. The typical "U" shape of test loss in this paper does not conflict with double descent, because we are plotting the weight norm instead of the number of model parameters [7]. However, the "U"-shape should better be considered as empirically common rather than provably universal. In fact, the interaction between properties of data and inductive biases of learning algorithms can be more complicated than double descent [18,19].

Initialization From the optimization perspective, initializations are usually based on the "edge of chaos" idea such that variance of features and gradients should be preserved in the forward and backward pass [9,20–23], or based on analyzing Jacobians and/or Hessians [24]. From the generalization perspective, it was shown that large initializations overfit data easily but result in poor generalization [25,26], which agrees with our LU mechanism.

Weight decay regularization is a standard trick in machine learning and has various effects on optimization and generalization [27,28]. In particular, [29] observes that it takes $t \propto 1/\lambda$ training steps to reach maximum test performance. This is strikingly similar to the grokking time $t \propto 1/\lambda$ we derived from the LU mechanism.

6.7 Conclusions

This study elucidates the grokking phenomenon from the perspective of loss landscapes. Our conclusions are: (i) grokking originates from the mismatch between training and test losses ("LU" mechanism). (ii) grokking can happen in various models for a wide range of datasets, although the grokking signature is usually most dramatic for algorithmic datasets. (iii) The dramaticness of grokking depends on how much the task relies on learning representations. This work not only reveals the mechanism of grokking, but also shows that reduced landscape analysis is a useful tool for characterizing data-model interaction and representation learning.

Acknowledgements

We thank Wenxian Shi, Niklas Nolte, Ouail Kitouni and Mike Williams for helpful discussions. This work was supported by The Casey and Family Foundation, the Foundational Questions Institute, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant

No. 2141064), and the NSF AI Institute for Artificial Intelligence and Fundamental Interactions (IAIFI) through NSF Grant No. PHY-2019786.

References

- [1] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra. “Grokking: Generalization beyond overfitting on small algorithmic datasets.” *arXiv preprint arXiv:2201.02177*, 2022.
- [2] Z. Liu, O. Kitouni, N. Nolte, E. J. Michaud, M. Tegmark, and M. Williams. “Towards Understanding Grokking: An Effective Theory of Representation Learning.” *arXiv preprint arXiv:2205.10343*, 2022.
- [3] V. Thilak, E. Littwin, S. Zhai, O. Saremi, R. Paiss, and J. Susskind. “The Slingshot Mechanism: An Empirical Study of Adaptive Optimizers and the Grokking Phenomenon.” *arXiv preprint arXiv:2206.04817*, 2022.
- [4] B. Barak, B. L. Edelman, S. Goel, S. Kakade, E. Malach, and C. Zhang. “Hidden Progress in Deep Learning: SGD Learns Parities Near the Computational Limit.” *arXiv preprint arXiv:2207.08799*, 2022.
- [5] S. Fort and A. Scherlis. “The goldilocks zone: Towards better understanding of neural network loss landscapes.” In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 3574–3581.
- [6] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak, and I. Sutskever. “Deep double descent: Where bigger models and more data hurt.” *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12), 2021, p. 124003.
- [7] A. Ng and T. Ma. *CS229 Lecture Notes*. 2022. URL: https://cs229.stanford.edu/lectures-spring2022/main%5C_notes.pdf.
- [8] S. S. Schoenholz, J. Gilmer, S. Ganguli, and J. Sohl-Dickstein. “Deep information propagation.” *arXiv preprint arXiv:1611.01232*, 2016.
- [9] G. Yang and S. Schoenholz. “Mean field residual networks: On the edge of chaos.” *Advances in neural information processing systems*, 30, 2017.
- [10] L. Deng. “The mnist database of handwritten digit images for machine learning research.” *IEEE Signal Processing Magazine*, 29(6), 2012, pp. 141–142.
- [11] S. Hochreiter and J. Schmidhuber. “Long short-term memory.” *Neural computation*, 9(8), 1997, pp. 1735–1780.
- [12] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. “Learning Word Vectors for Sentiment Analysis.” In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. URL: <http://www.aclweb.org/anthology/P11-1015>.
- [13] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld. “Quantum chemistry structures and properties of 134 kilo molecules.” *Scientific Data*, 1, 2014.

- [14] N. Nanda, L. Chan, T. Liberum, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.
- [15] R. Shah. *Alignment Newsletter #159*. 2021. URL: https://www.alignmentforum.org/posts/zvWqPmQasssaAWkrj/an-159-building-agents-that-know-how-to-experiment-by%5C#DEEP_LEARNING_.
- [16] B. Millidge. *Grokking ‘grokking’*. <https://beren.io/2022-01-11-Grokking-Grokking/>. 2022.
- [17] F. F. Yilmaz and R. Heckel. “Regularization-wise double descent: Why it occurs and how to eliminate it.” In: *2022 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2022, pp. 426–431.
- [18] L. Chen, Y. Min, M. Belkin, and A. Karbasi. “Multiple descent: Design your own generalization curve.” *Advances in Neural Information Processing Systems*, 34, 2021, pp. 8898–8912.
- [19] S. d’Ascoli, L. Sagun, and G. Biroli. “Triple descent and the two kinds of overfitting: Where & why do they appear?” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 3058–3069.
- [20] X. Glorot and Y. Bengio. “Understanding the difficulty of training deep feedforward neural networks.” In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.
- [21] K. He, X. Zhang, S. Ren, and J. Sun. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.” In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.
- [22] Y. Bahri, J. Kadmon, J. Pennington, S. S. Schoenholz, J. Sohl-Dickstein, and S. Ganguli. “Statistical mechanics of deep learning.” *Annual Review of Condensed Matter Physics*, 11(1), 2020.
- [23] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Soljačić. “Tunable efficient unitary neural networks (eunn) and their application to rnns.” In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1733–1741.
- [24] M. Skorski, A. Temperoni, and M. Theobald. “Revisiting Initialization of Neural Networks.” *arXiv preprint arXiv:2004.09506*, 2020.
- [25] Z.-Q. J. Xu, Y. Zhang, and Y. Xiao. “Training behavior of deep neural network in frequency domain.” In: *International Conference on Neural Information Processing*. Springer. 2019, pp. 264–274.
- [26] Y. Zhang, Z.-Q. J. Xu, T. Luo, and Z. Ma. “A type of generalization error induced by initialization in deep neural networks.” In: *Mathematical and Scientific Machine Learning*. PMLR. 2020, pp. 144–164.
- [27] G. Zhang, C. Wang, B. Xu, and R. Grosse. “Three mechanisms of weight decay regularization.” *arXiv preprint arXiv:1810.12281*, 2018.
- [28] T. Van Laarhoven. “L2 regularization versus batch and weight normalization.” *arXiv preprint arXiv:1706.05350*, 2017.
- [29] A. Lewkowycz and G. Gur-Ari. “On the training dynamics of deep networks with L_2 regularization.” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 4790–4799.

- [30] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization.” *arXiv preprint arXiv:1412.6980*, 2014.
- [31] V. Varma, R. Shah, Z. Kenton, J. Kramár, and R. Kumar. “Explaining grokking through circuit efficiency.” *arXiv preprint arXiv:2309.02390*, 2023.

6.8 Appendix

6.8.1 Experiment details

Sentiment analysis of text IMDb [12] includes 50k movie reviews to be classified as being positive or negative. To pre-process the data, we extract the 1000 most frequent words and tokenize each review into an array of token indices. Less frequent words are ignored, and each review array is padded to length 500. We adopt the LSTM model [11] to perform the classification, with two layers, embedding dimension 64, and hidden dimension 128. We use the Adam optimizer [30] with learning rate 0.001 to minimize the binary cross entropy loss. We hold back 25% of the dataset for testing.

Molecules QM9 is a database for small molecules and their properties. We use a graph convolutional neural network (GCNN) to predict the isotropic polarizability. The GCNN contains 2 convolutional layers with ReLU activation, followed by a linear layer. We use the Adam optimizer with learning rate 0.001 to minimize the MSE loss. We split the dataset into 50/50 train/test.

MNIST We train width-200 depth-3 ReLU MLPs on the MNIST dataset with MSE loss. We use the AdamW optimizer with a learning rate of 0.001 and a batch size of 200.

6.8.2 Reduced loss for modular addition with transformers

In Figure 6.9 we show reduced loss landscape plots for transformers trained on modular addition. We use the setup of [14] and train a 1-layer transformer on modular addition ($p = 113$) with $d_{\text{model}} = 128$, 4 attention heads, and $d_{\text{mlp}} = 512$ with ReLU activations. We train with a learning rate of 0.001 while constraining model weight norm, for a variety of α and a variety of train set fractions. The LU shape holds for $\alpha \in [0.1, 4]$ (some optimization issue may be responsible for the rise in train loss for $\alpha > 4$). We see the critical train set size is approximately 0.25, in line with earlier studies on grokking.

6.8.3 time to generalize versus weight decay

In our discussion of the “LU mechanism” as an explanation for grokking in Section 6.2, we predicted that the training time required for a model to generalize should be $t \propto \gamma^{-1}$ where γ is the weight decay. To test this, we perform a grid search over weight decays γ and plot the number of training steps required for models to reach a specified level of test accuracy in Figure 6.10a-6.10b. We also show full training curves for these runs in Figure 6.10c-6.10d. We perform experiments in two setups:

- (a) **Transformer on modular addition:** We use the replication of grokking from Nanda et al. [14] and train a 1-layer transformer on modular addition ($p = 113$ and a train set fraction

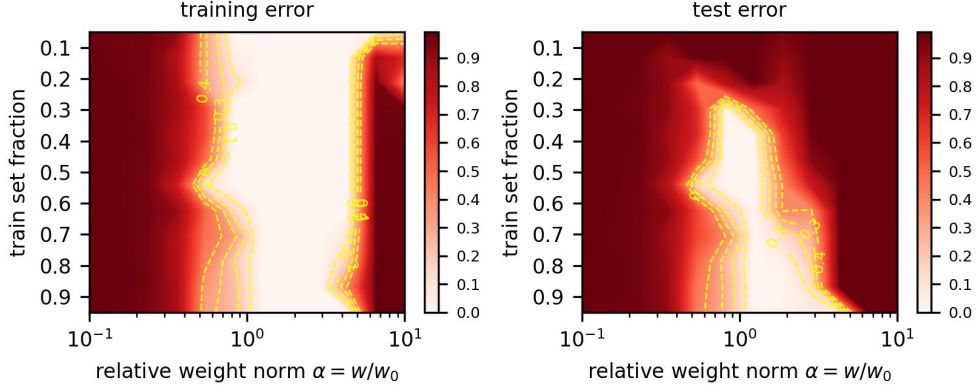


Figure 6.9: Reduced loss landscapes for transformers trained on modular addition, the original setting where grokking was observed.

of 0.3) where $d_{\text{model}} = 128$, with 4 attention heads, $d_{\text{mlp}} = 512$, ReLU activations, and an AdamW learning rate of 0.001. From Figure 6.10a, we find that $t \propto \gamma^{-1}$ holds across roughly two orders of magnitude of t and γ . There is some seed dependence on the generalization time (some seeds consistently require longer to generalize), but for each seed (corresponding to a particular model initialization) the relation $t \propto \gamma^{-1}$ appears to fit the data well.

- (b) **ReLU MLP on MNIST:** We train ReLU MLPs on MNIST as described in Appendix 6.8.1. We use an $\alpha = 9.0$ and train on a reduced training set of 1000 samples to delay generalization / induce grokking. From Figure 6.10b, we find that for γ roughly between 0.1 and 1.0 the relation $t \propto \gamma^{-1}$ holds. Very high values of weight decay seem to mess with optimization. On the other hand, with very low weight decay the model generalizes faster than naively expected, perhaps due to implicit regularization.

6.8.4 Section 6.5.1 setup

Architecture Similar to [2], the decoder architecture is an MLP with hard coded addition. Each input symbol i is encoded to a scalar E_i . Each output symbol k is represented by a 30D random vector $\hat{\mathbf{Y}}_k$. We consider addition with base p , so input $0 \leq i, j \leq p - 1$ and output $0 \leq k = i + j \leq 2(p - 1)$. We denote *representation* as $\mathbf{R} = \{E_0, E_1 \dots, E_{p-1}\}$. The MLP has two hidden layers, with neurons 1-200-200-30 in each layer and ReLU activations. Given a training sample $(E_i, E_j) \rightarrow \mathbf{Y}_k$ where $i + j = k$, the prediction of the MLP decoder is

$$\mathbf{Y}_k = \text{Dec}_{\mathbf{w}}(E_i + E_j), \quad (6.8)$$

and the loss function being the mean squared error (MSE) between $\mathbf{Y}_k$ and $\hat{\mathbf{Y}}_k$, and $\mathbf{w}$ being the decoder weight. Although the common setup of grokking is to make both the representation $\mathbf{R}$ and the decoder $\mathbf{w}$ trainable, we will freeze part of them for easier analysis. This is where it could be a bit confusing, so we explicitly distinguish three setups: *landscape analysis*, *reduced trajectory analysis* and *full trajectory analysis*. Each setup have different subset of trainable parameters, as shown in Table 6.1.

Trainability	decoder weight $\mathbf{w}$		Representation $\mathbf{R}$	
	norm $w = \ \mathbf{w}\ _2$	direction $\hat{\mathbf{w}} = \mathbf{w}/w$	messiness m	Other
Landscape analysis	No, w	Yes	No, m	No, 0
Reduced trajectory	Yes	No, $\hat{\mathbf{w}}^*(w, m)$	Yes	No, 0
Full trajectory	Yes	Yes	Yes	Yes

Table 6.1: Threes setups used in this paper, with different set of parameters trainable.

Landscape analysis Both the representation $\mathbf{R}$ and weight norm w are fixed. Only the weight direction $\hat{\mathbf{w}}$ is trainable. The representation $\mathbf{R}$ is fixed according to Eq. (6.4), which is dependent on m , the representation messiness. The decoder has fixed weight norm w , but the weight direction $\hat{\mathbf{w}}$ is trainable. For each fixed (w, m) , we minimize training loss over $\hat{\mathbf{w}}$ to get

$$\hat{\mathbf{w}}^*(w, m) = \underset{\hat{\mathbf{w}}}{\operatorname{argmin}} \ell_{\text{train}}(w, m, \hat{\mathbf{w}}), \quad (6.9)$$

and define reduced training and test loss, as in Eq. (6.5). The minimization is implemented by the Adam optimizer with learning rate 10^{-3} for 10^4 steps. Although (w, m) are not trainable, we repeat the above minimization independently for different (w, m) . In Figure 6.6 (a)(b)(d), the background heatmaps belong to landscape analysis.

Reduced trajectory analysis is a “thought experiment” based on landscape analysis. Since full trajectory analysis can be intractable due to too high dimensions, we try to reduce the trajectory analysis to 2D, by making two assumptions about the real dynamics: (1) *Scale separation*: the dynamics of $\hat{\mathbf{w}}$ is much faster than the dynamics along w and along m , such that $\hat{\mathbf{w}}(t) = \hat{\mathbf{w}}^*(w(t), m(t))$ is valid at every moment during training. (2) *Representation evolution is linear*, i.e., interpolating between initial random Gaussian and final linear representation. With these two assumptions, the training dynamics is effectively reduced to 2D, depending only on (w, m) , obeying Eq. (6.6). In Figure 6.6 (a)(b)(c), the path $A \rightarrow E$ belongs to reduced trajectory analysis.

Admittedly the reduced trajectory may deviate from the full trajectory since the assumptions may not be met, but it can shed light on the full trajectory: the weight norm first increases and then increases, and the decrease of weight norm is highly correlated with generalization (please Figure 6.7).

6.8.5 MNIST experiments with cross entropy loss

To respond to a reviewer’s concern that our use of the MSE loss is the “secret” to get grokking on MNIST (Figure 6.3), we reran our experiments with the cross entropy (CE) loss. The results are qualitatively similar, with some quantitative differences.

Landscape analysis

Comparing Figure 6.3 (MSE) and Figure 6.11 (CE), we notice the they are qualitatively similar: (1) for small datasets, the reduced training error and test error resemble an “L” and “U” against the weight norm, respectively; (2) for large datasets, the “U” becomes more like “L”, i.e., the mismatch between the reduced training and test error is small. However, a quantitative difference exist: CE produces a broader “Goldilocks zone” (the weight range where generalization happens) than MSE. This implies that to induce grokking with CE, we need to increase the weight norm to a larger value (say $\alpha = 100$).

Training dynamics

We are able to observe delayed generalization during training on MNIST with cross entropy loss, but doing so requires a higher α than was necessary when using MSE loss, as predicted by the reduced loss landscapes in Figure 6.11. Figure 6.12 shows training trajectories from a 3-layer ReLU MLP on MNIST trained with cross entropy loss with $\alpha = 100$ and $D = 200$. We see that test accuracy rises to 30-40% early in training, then plateaus for an extended period, before increasing to $\approx 75\%$ while train accuracy remains at 100%. While the dynamics are not as clean as with MSE loss, since test accuracy first plateaus at better-than-random accuracy, we think it is still fair to classify these dynamics as “grokking” due to the improvement in generalization late in training after a plateau.

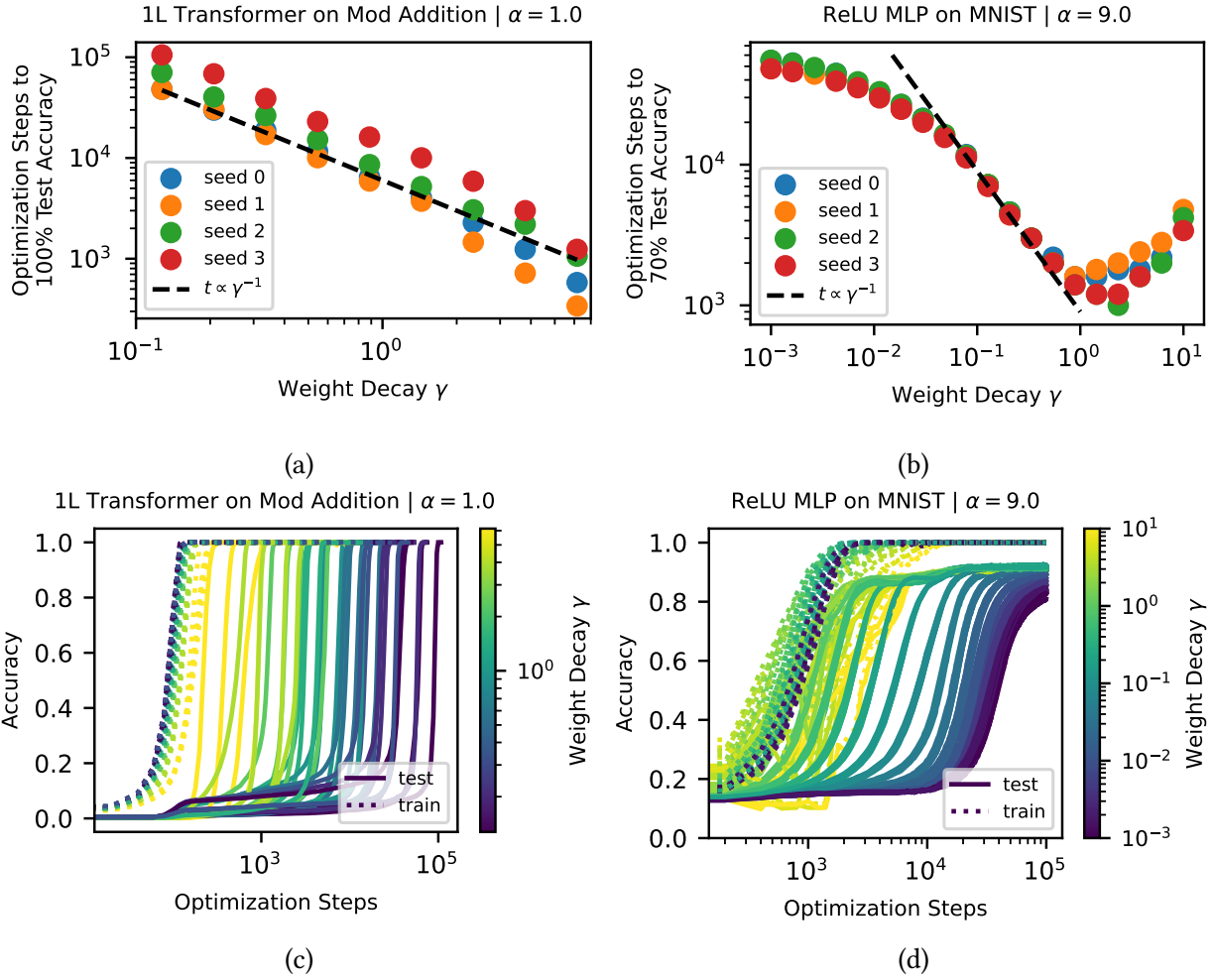


Figure 6.10: Time to generalize as a function of weight decay: we investigate to what extent the relation $t \propto \gamma^{-1}$ holds, where t is number of training steps needed for the model to generalize and γ is the AdamW weight decay. When a lower weight decay is used, models spend longer in the period of overfitting before eventually generalizing. We show the generalization time t as a function of γ in (a)-(b) and full training curves for these runs in (c)-(d).

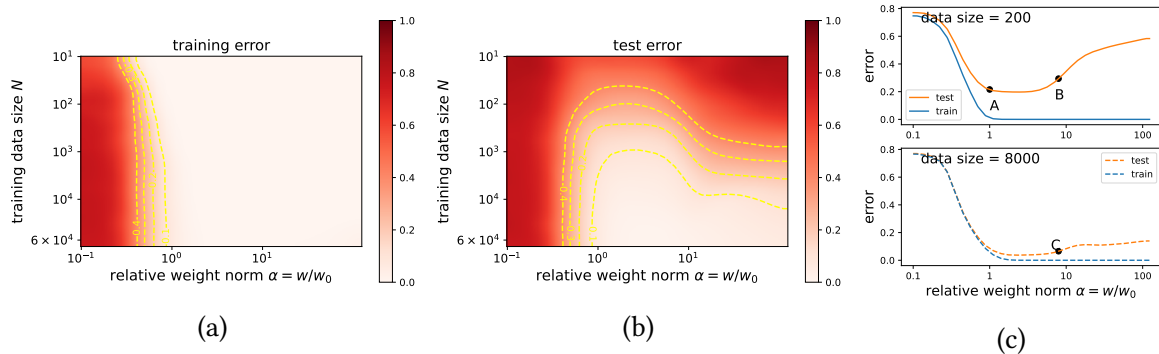


Figure 6.11: MNIST with the cross entropy loss (as opposed to the MSE loss used in Figure 6.3). (a) reduced training error, (b) reduced test error. (c) "LU" still holds for the cross entropy loss, but the effect is milder than the MSE loss. In particular, the "Goldilocks zone" (the weight range where generalization happens) is broader.

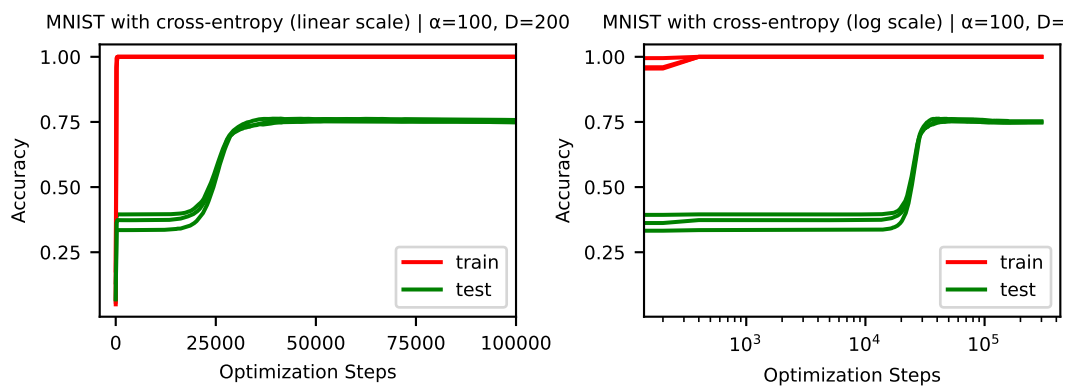


Figure 6.12: Training curves using cross entropy loss on MNIST. We are still able to observe delayed generalization on MNIST using cross entropy loss, though test accuracy first plateaus at higher than random-guess accuracy.

The modern synthesis on grokking

In the past two chapters, we have seen two explanations of grokking from papers written in 2022. Since then, there has been a large amount of additional work on grokking from the research community. In this short note, I will describe a more up-to-date theory of grokking from the literature which synthesizes aspects from both of our two papers and rebuts other aspects.

In Chapter 5, we studied how structured representations are needed for generalization on certain tasks. Our basic idea was that structured representations take longer to learn. At first, networks memorize where to map unstructured representations, but slowly the representations form structure, which enables a solution where the network implements a more sophisticated algorithm using those representations and which generalizes.

In Chapter 6, we studied how the weight norm of neural networks impacts generalization. We observed that networks with high weight norm tend to generalize worse than networks with low weight norm, and that the weight norm drops as networks generalize in the setting where grokking occurs.

One highly impactful paper on grokking from other authors was Neel Nanda’s “Progress measures for grokking via mechanistic interpretability”[14], based on a blog post Neel published in August 2022. This work did not just study the structured circular representations that we found, but described how those representations are used by an end-to-end circuit that implements modular addition. This algorithm essentially uses trigonometric identities to add the angles of the representation of the two inputs on the circle. Neel found that this circuitry forms *before* the network begins to generalize, suggesting that generalization happens when networks clean up the “memorisation circuit” that had formed early in training. From this perspective, our observation that the circular representations appearing in the PCA axes coincided with generalization was likely due to the “clean up” phase removing noise in other dimensions, causing the PCA axes to align with the circular representations which were already there. Neel suggested that grokking occurs because the “generalising circuit” takes longer to form during training, essentially because it is more complex than the “memorising circuit”, and that it has lower weight norm than the “memorising circuit”. Since networks are trained with weight decay, once the slower-forming “generalising circuit” is present, the “memorising circuit” is removed by the network in order to reduce its weight norm.

The ideas of Neel’s work were made more rigorous in Varma & Shah et al. “Explaining grokking through circuit efficiency.” This work observed directly that the weight norm of the “memorising circuit” (holding training set loss fixed) increases linearly with the size of the training set, whereas the “generalising circuit” has constant weight norm.

... continued

Varma & Shah note that the question of “What causes generalising circuits to develop slower? [than the memorising circuit]” is left open by their work. Here, I’ll suggest an answer, which is implied, though not stated exactly, by Neel’s work: **The memorizing circuit is learned faster than the generalizing circuit because it is not a single circuit, but rather a large number of very simple, independent mechanisms which memorize individual points in the training set.** Since these mechanisms are independent, and each have the same complexity, they are learned at roughly the same time, but this does not mean that there is a true unified “memorization circuit”. The generalizing circuit forms more slowly because it is much more complex than the mechanisms which memorize individual datapoints. However, the total weight norm required to implement the collection of memorizing mechanisms is greater than the weight norm required to implement the generalizing circuit.

The unified perspective (largely paraphrasing Nanda): When we train networks on the “algorithmic” tasks where grokking happens, in order for networks to generalize, they must learn a fairly complex mechanism. This mechanism requires highly structured representations to be learned as part of a sophisticated algorithm. Because this mechanism is complex, it takes many steps of training to develop. The network can also learn a mechanism which memorizes each individual sample in the training set. These individual mechanisms are very simple, and are learned in parallel very quickly in training. There are as many individual memorization mechanisms as there are samples in the training set. While the more complex, generalizing mechanism takes longer to form, it also requires lower weight norm than the total weight norm of all memorizing mechanisms (when the training set is sufficiently large). Eventually, when the more complex generalizing mechanisms forms, the network cleans up all the memorizing mechanisms, lowering the weight norm, and leaving behind just the generalizing mechanism. Grokking occurs then due to differences in the complexity and efficiency of different possible mechanisms that neural networks can learn.

Chapter 7

Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code^{*}

Many assume that neural networks are irreducible black boxes, and cannot be given any description other than the inscrutable weights of the network itself θ . In this work, we show that this is not the case. We can take recurrent neural networks, trained on algorithmic/mathematical tasks, and decode their internal representations and computation into functioning, standalone, interpretable Python programs that replicate the functioning of the network.

Abstract

Can we turn AI black boxes into code? Although this mission sounds extremely challenging, we show that it is not entirely impossible by presenting a proof-of-concept method, MIPS, which can synthesize programs 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 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.

^{*}Eric J. Michaud, Isaac Liao, Vedang Lad, Ziming Liu, Anish Mudide, Chloe Loughridge, Zifan Carl Guo, Tara Rezaei Kheirkhah, Mateja Vukelić, and Max Tegmark. “Opening the AI black box: Distilling machine-learned algorithms into code.” *Entropy* 26, no. 12 (2024): 1046.

7.1 Introduction

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

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

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

Our paper aims to inject optimism into this field by taking a small but nontrivial step toward showing that automatic mechanistic interpretability (AutoMI) is not impossible, where AutoMI means black-box models can be turned into programs automatically without human inspection. Specifically, we present a proof-of-concept method MIPS (Mechanistic-Interpretability-based Program Synthesis), which can distill simple learned algorithms from neural networks into Python code, for small-scale algorithmic tasks. The main goal of this paper is not to present a method that fully solves AutoMI, but to demonstrate progress towards AutoMI with a simple proof of concept. The rest of this paper is organized as follows. After reviewing prior work in Section 7.2, we present our method in Section 7.3, test it on a benchmark in Section 7.4 and summarize our conclusions in Section 7.5.

7.2 Related Work

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

Our work builds on the recent progress in *mechanistic interpretability* (MI) of neural networks [8–11]. Much MI work has tried to understand how neural networks represent various types of information, e.g., geographic information [12–14], truth [15,16] and the state of board

games [17–19]. Another major MI thrust has been to understand how neural networks perform algorithmic tasks, e.g., modular arithmetic [20–23] and other group operations [24], greater-than [25], and greatest-common-divisor [26]. The key step of mechanistic interpretability is to look into discovering structures in hidden representations. In this paper, we manage to discover bit representations, integer representations, and clusters (finite state machines).

Whereas Lindner et al. [27] automatically convert traditional code into a neural network, we aim to do the opposite, as was also recently demonstrated in Friedman, Wettig, and Chen [28]. One direction in automating mechanistic interpretability uses LLMs to label internal units of neural networks such as neurons [29] and features discovered by sparse autoencoders [30,31].

Another recent effort in automating MI involves automatically identifying which internal units causally influence each other and the network output for a given set of inputs, [32–34]. However, these methods do not automatically generate pseudocode or give a description of *how* the states of downstream units are computed from upstream units, which we aim to do in this work.

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

7.3 MIPS, our program synthesis algorithm

As summarized in Figure 7.1, MIPS involves the following key steps.

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

$$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_i), \quad (7.1)$$

$$\mathbf{y}_i = g(\mathbf{h}_i), \quad (7.2)$$

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

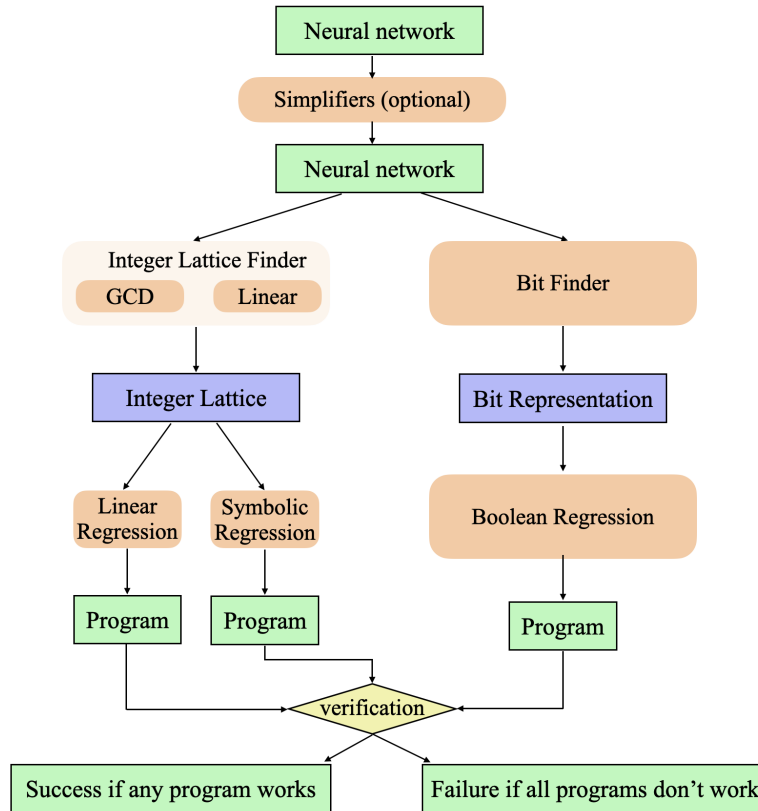


Figure 7.1: The pipeline of our program synthesis method. MIPS relies on discovering integer representations and bit representations of hidden states, which enable regression methods to figure out the exact symbolic relations between input, hidden, and output states.

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.

7.3.1 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 $n \in \{1, 2, \dots, 128\}$, $w_f \in \{1, 2, \dots, 256\}$, $d_f \in \{1, 2, 3\}$, $w_g \in \{1, 2, \dots, 256\}$ 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) $\mathbf{p}_i$ 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.

7.3.2 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:

$$\begin{aligned}\mathbf{h}'_i &= \mathbf{A}\mathbf{h}_i = \mathbf{A}\mathbf{W}\mathbf{A}^{-1}\mathbf{A}\mathbf{h}_{i-1} + \mathbf{A}\mathbf{V}\mathbf{x}_i + \mathbf{A}\mathbf{b} \\ &= \mathbf{W}'^{-1}\mathbf{h}'_{i-1} + \mathbf{V}'\mathbf{x}_i + \mathbf{b}',\end{aligned}\tag{7.3}$$

$$\begin{aligned}\mathbf{y}_i &= G(\mathbf{U}\mathbf{h}_i + \mathbf{c}) = G(\mathbf{U}\mathbf{A}^{-1}\mathbf{h}'_i + \mathbf{c}) \\ &= G(\mathbf{U}'\mathbf{h}'_i + \mathbf{c}).\end{aligned}\tag{7.4}$$

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 7.6.7.

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.

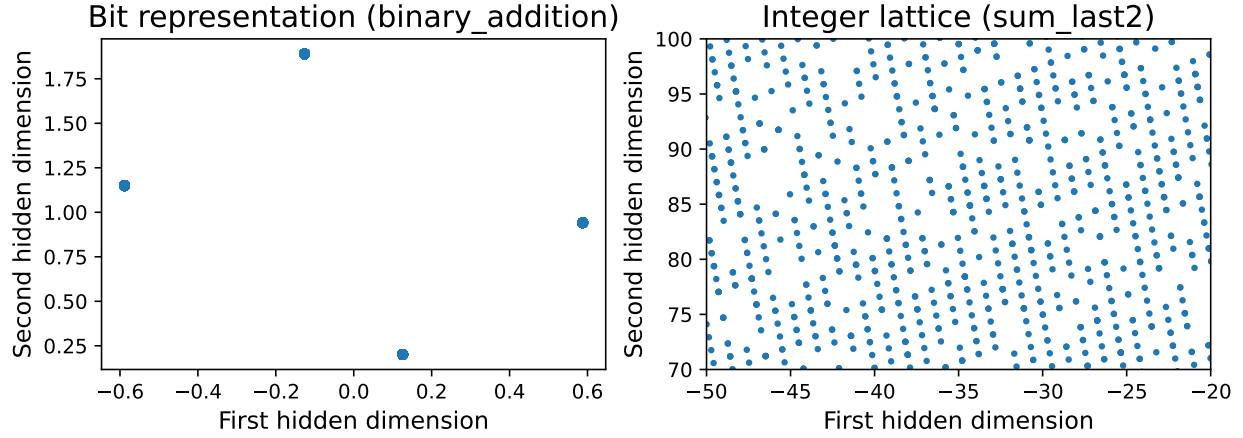


Figure 7.2: These hidden structures can be turned into discrete representations. Left: bitstring addition, corresponding to 2 bits: the output bit and the carry bit. Right: Sum_Last2 task, 2D lattice corresponding to two integers.

3. **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 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.

7.3.3 Boolean and integer autoencoders

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

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

create an automated method that distills the learned representation and associated algorithms into an equivalent (Python) program?

Since the tasks in our benchmark involve bits and integers, which are already discrete, the only non-discrete parts in a recurrent neural network are its hidden representations. Here we show two cases when hidden states can be discretized: they are (1) a bit representation or (2) a (typically incomplete) integer lattice. Generalizing to the mixed case of bits and integers is straightforward. Figure 7.2 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 7.2 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 7.2, 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 tuples 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 7.6.1, 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 $\{1.7, 3.2, 6.2, 7.7\dots\}$, 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 7.6.5. 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* [42], including the 6 unary operations $\{>, <, \sim, H, D, A\}$ and 4 binary operations $\{+, -, *, \%\}$ whose meanings are explained in Appendix 7.6.6, 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 7.3 and 7.4 and in Appendix 7.6.9. Most hyperparameters are thresholds. For example, when a lattice has the reconstruction error ϵ below some threshold θ , we claim an integer lattice has been detected. Since we define the reconstruction errors to be dimensionless, it is clear that $\epsilon = 1$ means no structure detected and $\epsilon = 0$ means perfect integer lattices. We find our results to be fairly robust with respect to θ , e.g., $\theta = 10^{-2} - 10^{-1}$ would yield the same results.

7.4 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.

7.4.1 Benchmark

Our benchmark consists of the 62 algorithmic tasks listed in Table 7.1. 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 7.1, with additional details in Appendix 7.6.8. The benchmark aims to cover a diverse range of algorithmic tasks. To balance between different families, when a group of tasks is similar (e.g., summing up the last k bits), our convention is to keep (at most) six of them.

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

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

7.4.2 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 7.5.

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 7.1: Benchmark results. For tasks with note “see text”, please refer to Appendix 7.6.8. The last column highlights the MIPS module responsible for each task. BR = boolean regression, LR = linear regression, SR = symbolic regression, and NA means MIPS is not expected to solve this problem. Green means success and red means failure.

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

```

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

```

Figure 7.3: 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.

7.4.3 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 7.3.1 discovers networks of varying task-dependent shape and size. Table 7.2 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 7.3 shows an example of a MIPS rediscovering the “ripple-carry adder” algorithm. The normalizers significantly simplified some of the resulting programs, as illustrated in Figure 7.4, 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.

```

1 def f(s):
2     a = 198;b = -11;c = -3;d = 483;e = 0;
3     ys = []
4     for i in range(20):
5         x = s[i]
6         next_a = -b+c+190
7         next_b = b-c-d-e+x+480
8         next_c = b-e+8
9         next_d = -b+e-x+472
10        next_e = a+b-e-187
11        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
12        y = -d+483
13        ys.append(y)
14    return ys

```

```

1 def f(s):
2     a = 0;b = 0;c = 0;d = 0;e = 0;
3     ys = []
4     for i in range(20):
5         x = s[i]
6         next_a = +x
7         next_b = a
8         next_c = b
9         next_d = c
10        next_e = d
11        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
12        y = a+b+c+d+e
13        ys.append(y)
14    return ys

```

Figure 7.4: Comparison of code generated from an RNN trained on Sum_Last5, without (top) and with (bottom) normalizers.

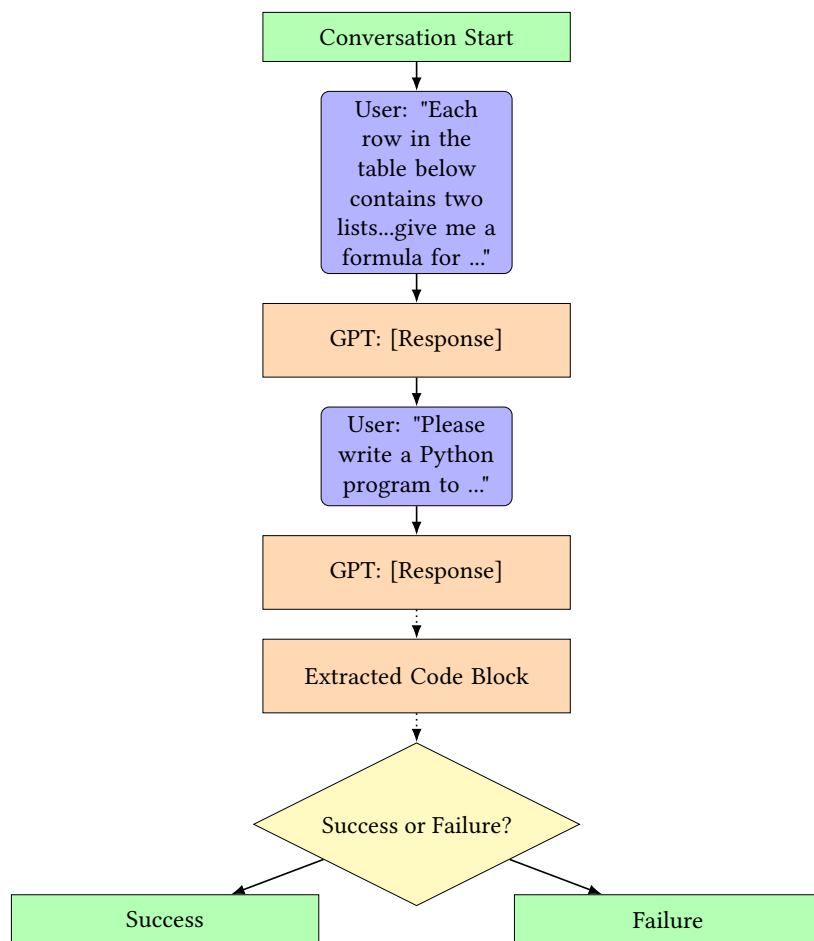


Figure 7.5: We compare MIPS against program synthesis with the large language model GPT-4 Turbo, prompted with a “chain-of-thought” approach. It begins with the user providing a task, followed by the model’s response, and culminates in assessing the success or failure of the generated Python code based on its accuracy in processing the provided lists.

7.5 Discussion

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.

7.5.1 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. Please note that our motivation is not to outcompete other program synthesis methods, but instead to provide a proof-of-principle that fully automated distillation of machine-learned algorithms is not impossible.

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

Program synthesis aside, our results shed further light on mechanistic interpretability, specifically on how neural networks represent bits and integers. We found that n integers tend to get encoded linearly in n dimensions, but generically in non-orthogonal directions with an additive offset. This is presumably because there are many more such messy encodings than simple ones, and the messiness can be easily (linearly) decoded. We saw that n bits sometimes get encoded as an n -dimensional parallelogram, but not always --- possibly because linear decodability is less helpful when the subsequent bit operations to be performed are nonlinear anyway.

7.5.2 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 7.6.10 to show that our MIPS-learned ripple adder correctly adds *any* binary numbers, not merely those in the test set, but such manual work should ideally be fully automated)

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

Acknowledgements

We thank Wes Gurnee, James Liu, and Armaun Sanayei for helpful conversations and suggestions. This research was funded by Erik Otto, Jaan Tallinn, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), and IAIFI through NSF grant PHY-2019786.

References

- [1] Center for AI Safety. *Statement on AI Risk*. <https://www.safe.ai/work/statement-on-ai-risk>. Open Letter. Accessed: September 4, 2024. Center for AI Safety, 2023. URL: <https://www.safe.ai/work/statement-on-ai-risk>.
- [2] M. Tegmark and S. Omohundro. “Provably safe systems: the only path to controllable agi.” *arXiv preprint arXiv:2309.01933*, 2023.
- [3] D. Dalrymple, J. Skalse, Y. Bengio, S. Russell, M. Tegmark, S. Seshia, S. Omohundro, C. Szegedy, B. Goldhaber, N. Ammann, et al. “Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems.” *arXiv preprint arXiv:2405.06624*, 2024.
- [4] B. Zhou and G. Ding. “Survey of intelligent program synthesis techniques.” In: *International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2023)*. Vol. 12941. SPIE. 2023, pp. 1122–1136.
- [5] A. Odena, K. Shi, D. Bieber, R. Singh, C. Sutton, and H. Dai. “BUSTLE: Bottom-Up program synthesis through learning-guided exploration.” *arXiv preprint arXiv:2007.14381*, 2020.
- [6] J. Wu, L. Wei, Y. Jiang, S.-C. Cheung, L. Ren, and C. Xu. “Programming by Example Made Easy.” *ACM Transactions on Software Engineering and Methodology*, 33(1), 2023, pp. 1–36.
- [7] D. Sobania, M. Briesch, and F. Rothlauf. “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*. 2022, pp. 1019–1027.
- [8] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. “Zoom in: An introduction to circuits.” *Distill*, 5(3), 2020, e00024–001.
- [9] N. Cammarata, G. Goh, S. Carter, L. Schubert, M. Petrov, and C. Olah. “Curve Detectors.” *Distill*, 2020. <https://distill.pub/2020/circuits/curve-detectors>. DOI: [10.23915/distill.00024.003](https://doi.org/10.23915/distill.00024.003).

- [10] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt. “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small.” *arXiv preprint arXiv:2211.00593*, 2022.
- [11] C. Olsson et al. “In-context Learning and Induction Heads.” *Transformer Circuits Thread*, 2022. <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [12] G. Goh, N. C. †, C. V. †, S. Carter, M. Petrov, L. Schubert, A. Radford, and C. Olah. “Multimodal Neurons in Artificial Neural Networks.” *Distill*, 2021. <https://distill.pub/2021/multimodal-neurons>. DOI: [10.23915/distill.00030](https://doi.org/10.23915/distill.00030).
- [13] W. Gurnee and M. Tegmark. “Language models represent space and time.” *arXiv preprint arXiv:2310.02207*, 2023.
- [14] K. Vafa, J. Y. Chen, J. Kleinberg, S. Mullainathan, and A. Rambachan. “Evaluating the World Model Implicit in a Generative Model.” *arXiv preprint arXiv:2406.03689*, 2024.
- [15] C. Burns, H. Ye, D. Klein, and J. Steinhardt. “Discovering latent knowledge in language models without supervision.” *arXiv preprint arXiv:2212.03827*, 2022.
- [16] S. Marks and M. Tegmark. “The geometry of truth: Emergent linear structure in large language model representations of true/false datasets.” *arXiv preprint arXiv:2310.06824*, 2023.
- [17] T. McGrath, A. Kapishnikov, N. Tomašev, A. Pearce, M. Wattenberg, D. Hassabis, B. Kim, U. Paquet, and V. Kramnik. “Acquisition of chess knowledge in alphazero.” *Proceedings of the National Academy of Sciences*, **119**(47), 2022, e2206625119.
- [18] S. Toshniwal, S. Wiseman, K. Livescu, and K. Gimpel. “Chess as a testbed for language model state tracking.” In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 10. 2022, pp. 11385–11393.
- [19] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. “Emergent world representations: Exploring a sequence model trained on a synthetic task.” *arXiv preprint arXiv:2210.13382*, 2022.
- [20] N. Nanda, L. Chan, T. Liberum, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.
- [21] Z. Liu, O. Kitouni, N. Nolte, E. J. Michaud, M. Tegmark, and M. Williams. “Towards Understanding Grokking: An Effective Theory of Representation Learning.” In: *Advances in Neural Information Processing Systems*. Ed. by A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho. 2022. URL: <https://openreview.net/forum?id=6at6rB3IZm>.
- [22] Z. Zhong, Z. Liu, M. Tegmark, and J. Andreas. “The clock and the pizza: Two stories in mechanistic explanation of neural networks.” *arXiv preprint arXiv:2306.17844*, 2023.
- [23] P. Quirke et al. “Understanding Addition in Transformers.” *arXiv preprint arXiv:2310.13121*, 2023.
- [24] B. Chughtai, L. Chan, and N. Nanda. “A Toy Model of Universality: Reverse Engineering how Networks Learn Group Operations.” In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, July 2023, pp. 6243–6267. URL: <https://proceedings.mlr.press/v202/chughtai23a.html>.

- [25] M. Hanna, O. Liu, and A. Variengien. “How does GPT-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model.” *arXiv preprint arXiv:2305.00586*, 2023.
- [26] F. Charton. “Can transformers learn the greatest common divisor?” *arXiv preprint arXiv:2308.15594*, 2023.
- [27] D. Lindner, J. Kramár, S. Farquhar, M. Rahtz, T. McGrath, and V. Mikulik. “Tracr: Compiled transformers as a laboratory for interpretability.” *arXiv preprint arXiv:2301.05062*, 2023.
- [28] D. Friedman, A. Wettig, and D. Chen. “Learning transformer programs.” *Advances in Neural Information Processing Systems*, 36, 2024.
- [29] S. Bills, N. Cammarata, D. Mossing, H. Tillman, L. Gao, G. Goh, I. Sutskever, J. Leike, J. Wu, and W. Saunders. *Language models can explain neurons in language models*. <https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html>. 2023.
- [30] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey. “Sparse autoencoders find highly interpretable features in language models.” *arXiv preprint arXiv:2309.08600*, 2023.
- [31] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. URL: <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [32] A. Conmy, A. N. Mavor-Parker, A. Lynch, S. Heimersheim, and A. Garriga-Alonso. “Towards automated circuit discovery for mechanistic interpretability.” *arXiv preprint arXiv:2304.14997*, 2023.
- [33] A. Syed, C. Rager, and A. Conmy. “Attribution Patching Outperforms Automated Circuit Discovery.” *arXiv preprint arXiv:2310.10348*, 2023.
- [34] S. Marks, C. Rager, E. J. Michaud, Y. Belinkov, D. Bau, and A. Mueller. “Sparse feature circuits: Discovering and editing interpretable causal graphs in language models.” *arXiv preprint arXiv:2403.19647*, 2024.
- [35] A. Karpathy, J. Johnson, and L. Fei-Fei. “Visualizing and understanding recurrent networks.” *arXiv preprint arXiv:1506.02078*, 2015.
- [36] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. “LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks.” *IEEE Transactions on Visualization and Computer Graphics*, 24(1), 2018, pp. 667–676. DOI: [10.1109/TVCG.2017.2744158](https://doi.org/10.1109/TVCG.2017.2744158).
- [37] C. L. Giles, B. G. Horne, and T. Lin. “Learning a class of large finite state machines with a recurrent neural network.” *Neural Networks*, 8(9), 1995, pp. 1359–1365.
- [38] Q. Wang, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles. “An empirical evaluation of rule extraction from recurrent neural networks.” *arXiv preprint arXiv:1709.10380*, 2017.
- [39] G. Weiss, Y. Goldberg, and E. Yahav. “Extracting automata from recurrent neural networks using queries and counterexamples.” In: *International Conference on Machine Learning*. PMLR, 2018, pp. 5247–5256.

- [40] C. Oliva and L. F. Lago-Fernández. “On the interpretation of recurrent neural networks as finite state machines.” In: *Artificial Neural Networks and Machine Learning–ICANN 2019: Theoretical Neural Computation: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part I* 28. Springer. 2019, pp. 312–323.
- [41] E. Muškardin, B. K. Aichernig, I. Pill, and M. Tappler. “Learning finite state models from recurrent neural networks.” In: *International Conference on Integrated Formal Methods*. Springer. 2022, pp. 229–248.
- [42] S.-M. Udrescu, A. Tan, J. Feng, O. Neto, T. Wu, and M. Tegmark. “AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity.” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 4860–4871.
- [43] M. Cranmer. “Interpretable machine learning for science with PySR and SymbolicRegression.jl.” *arXiv preprint arXiv:2305.01582*, 2023.
- [44] M. Cranmer, A. Sanchez Gonzalez, P. Battaglia, R. Xu, K. Cranmer, D. Spergel, and S. Ho. “Discovering symbolic models from deep learning with inductive biases.” *Advances in Neural Information Processing Systems*, 33, 2020, pp. 17429–17442.
- [45] H. Ma, A. Narayanaswamy, P. Riley, and L. Li. “Evolving symbolic density functionals.” *Science Advances*, 8(36), 2022, eabq0279.
- [46] A. Gu and T. Dao. “Mamba: Linear-time sequence modeling with selective state spaces.” *arXiv preprint arXiv:2312.00752*, 2023.

7.6 Appendix

7.6.1 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.

7.6.2 Problem formulation

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

$$\mathbf{x}_j = \sum_{i=1}^D a_{ji} \mathbf{b}_i + \mathbf{c}, \quad (7.5)$$

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

Our problem: given N such data points $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N)$, 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 $\{\mathbf{b}_i \rightarrow \sum_{j=1}^D \Lambda_{ij} \mathbf{b}_j\}$ remains a solution if Λ 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 ($\mathbf{\Lambda} \in \mathbb{Z}^{D \times D}$, $\det(\mathbf{\Lambda}) = \pm 1$).

7.6.3 Regular GCD

As a reminder, given a list of n numbers $\{y_1, y_2, \dots, y_n\}$, 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 $\{d | y_i/d \in \mathbb{Z}, \text{ and the greatest common divisor (GCD) is the largest number in this set. Because}$

$$\text{GCD}(y_1, \dots, y_n) = \text{GCD}(y_1, \text{GCD}(y_2, \text{GCD}(y_3, \dots))), \quad (7.6)$$

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 k^{th} step, we perform division-with-remainder to find the quotient q_k and the remainder r_k so that $r_{k-2} = q_k r_{k-1} + r_k$ with $|r_{k-1}| > |r_k|$ 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, $\text{GCD}(55, 45) = 5$, because

$$\begin{aligned} 55 &= 1 \times 45 + 10, \\ 45 &= 4 \times 10 + 5, \\ 10 &= 2 \times 5 + 0. \end{aligned} \quad (7.7)$$

7.6.4 Generalized GCD in D dimensions

Given a list of n vectors $\{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n\}$ where $\mathbf{y}_i \in \mathbb{R}^D$, and assuming that these vectors are in the lattice described by Eq. (7.5), 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.,

$$\mathbf{q}_i = \sum_{j=1}^n m_{ij} \mathbf{y}_j, m_{ij} \in \mathbb{Z}, i = 1, 2, \dots, D. \quad (7.8)$$

The goal of D -dimensional GCD is to find a “minimal” parallelogram, such that its volume (which is $\det(\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_D)$) is GCD of volumes of other possible parallelograms. Once the minimal parallelogram is found², we can also determine $\mathbf{b}_i$ in Eq. (7.5), 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 $\mathbf{q}_i (i = 1, 2, \dots)$ whose volume is the unit volume.

Step 1: Compute unit volume V_0 . We first define *representative* parallelograms as one where all $i = 1, 2, \dots, D$, $\mathbf{m}_i \equiv (m_{i1}, m_{i2}, \dots, m_{iD})$ are one-hot vectors, i.e., with only one element

¹We 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.

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

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 $(\mathbf{y}_3, \dots, \mathbf{y}_{D+1})$, 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 :

$$\begin{aligned} V_1 &= \det(\mathbf{y}_1, \mathbf{y}_3, \mathbf{y}_4, \dots, \mathbf{y}_D) \\ V_2 &= \det(\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \dots, \mathbf{y}_D) \end{aligned} \quad (7.9)$$

Since

$$aV_1 + bV_2 = \det(a\mathbf{y}_1 + b\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \dots, \mathbf{y}_D), \quad (7.10)$$

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

$$V'_1, V'_2 = \text{GCD}(V_1, V_2), \quad (7.11)$$

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

$$\mathbf{y}'_1, \mathbf{y}'_2 = \text{GCD_with_predefined_quotients}(\mathbf{y}_1, \mathbf{y}_2). \quad (7.12)$$

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

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

$$0 = V'_2 = \det(\mathbf{y}'_2, \mathbf{y}_3, \mathbf{y}_4, \dots, \mathbf{y}_D), \quad (7.13)$$

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

$$\begin{pmatrix} 1 & p_{2 \rightarrow 1} & p_{3 \rightarrow 1} & \cdots & p_{D \rightarrow 1} \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \quad (7.14)$$

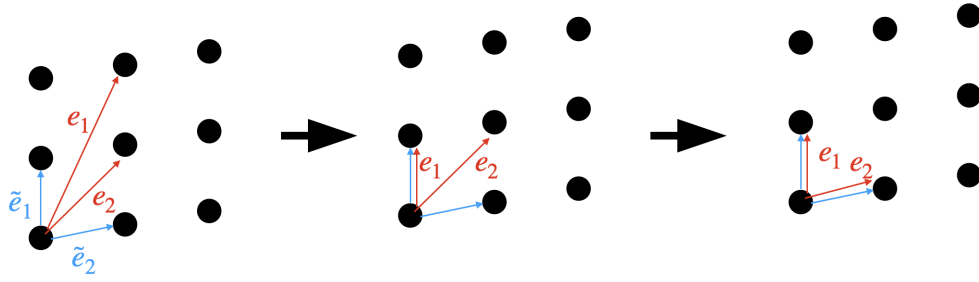


Figure 7.6: Both red and blue basis form a minimal parallelogram (in terms of cell volume), but one can further simplify red to blue by linear combination (simplicity in the sense of small ℓ_2 norm).

We do this iteratively, until no norm can become shorter via project-and-subtract. Please see Figure 7.6 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 ϵ_{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 ϵ_{gcd} and p depends on data and neural network training which we do not know a priori, we choose to grid sweep $\epsilon_{\text{gcd}} \in [10^{-3}, 1]$ and $p \in (0.1, 100)$; for each $(\epsilon_{\text{gcd}}, p)$, we obtain an integer lattice and compute its description length. We select the $(\epsilon_{\text{gcd}}, p)$ 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 $\log(1 + (\frac{AZ+b-X}{\epsilon_{\text{dl}}})^2)$ with $\epsilon_{\text{dl}} = 10^{-4}$.

7.6.5 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 $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(t)}$, then $\mathbf{h}^{(t)}$ is

$$\mathbf{h}^{(t)} = \sum_{j=1}^t W_h^{t-j} W_i \mathbf{x}_j, \quad (7.15)$$

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

$$W_h^{t-j} W_i, j = 1, 2, \dots, t, \quad (7.16)$$

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.

7.6.6 Symbolic regression

The formulation of symbolic regression is that one has data pair $(\mathbf{x}_i, y_i), i = 1, 2, \dots, N$ 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.

7.6.7 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 :

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

and is implemented by changing the weights:

$$\begin{aligned}\mathbf{W} &\Rightarrow \mathbf{A}\mathbf{W}\mathbf{A}^{-1} \\ \mathbf{V} &\Rightarrow \mathbf{A}\mathbf{V} \\ \mathbf{b} &\Rightarrow \mathbf{A}\mathbf{b} \\ \mathbf{U} &\Rightarrow \mathbf{U}\mathbf{A}^{-1}\end{aligned}$$

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 [46] 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.

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.

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:

$$\begin{pmatrix} 0 & 1 \\ \delta & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} \sqrt{\delta} & 0 \\ 0 & -\sqrt{\delta} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}^{-1} \quad (7.17)$$

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

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}^{-1} \quad (7.18)$$

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 (7.18) instead of from Equation (7.17).

Our algorithm first computes the eigenvalues λ_i , and then iteratively solves for the generalized eigenvectors which lie in $\ker((\mathbf{W} - \lambda_i \mathbf{I})^k)$ 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 $\mathbf{X} = \mathbf{V}_1 \mathbf{S} \mathbf{V}_2^H$ 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 $\epsilon\text{-ker}(\mathbf{W} - \lambda\mathbf{I})$. 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 $\mathbf{F}_0 = \ker(\mathbf{W} - \lambda\mathbf{I})$. Then for $k > 0$, we first solve for $\mathbf{J}_k = \epsilon\text{-ker}((\mathbf{I} - \text{proj}(\mathbf{F}_{k-1}))(\mathbf{W} - \lambda\mathbf{I}))$ 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

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

Toeplitz Transformation

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

De-biasing Transformation

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

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.

7.6.8 Supplementary training data details

Here we present additional details on the benchmark tasks marked "see text" in Table 7.1:

- **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 $F = input - 1, v \mapsto v + F, x \mapsto x + v$.
- **Newton_Spring:** This is an euler forward propagation technique which follows the equation $F = input - x, v \mapsto v + F, x \mapsto x + v$.
- **Newton_Magnetic:** This is an euler forward propagation technique which follows the equation $F_x = input_1 - v_y, F_y = input_2 + v_x, \mathbf{v} \mapsto \mathbf{v} + \mathbf{F}, \mathbf{x} \mapsto \mathbf{x} + \mathbf{v}$.

7.6.9 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 (
8         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
9         d) or (a and b and c and d)
10        next_b = c+d==0 or c+d==2
11        a = next_a;b = next_b;
12        y = a+b>1
13        ys.append(y)
14    return ys
```

Bitwise-Not

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

Parity-Last2

```
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 = c
8         next_b = a ^ c
9         a = next_a;b = next_b;
10        y = b
11        ys.append(y)
12    return ys
```

Parity-Last3

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

Parity-All

```
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
8         a = next_a;
9         y = a
10        ys.append(y)
11    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-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-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-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

```
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

```
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

```
1
2 def f(s):
3     a = 0;b = 295;c = 99;d = 0;e = 297;f = 99;
4     ys = []
5     for i in range(20):
6         x = s[i]
7         next_a = -b+295
8         next_b = b-c+f
9         next_c = b-c+d-97
10        next_d = -f+99
11        next_e = -a+297
12        next_f = -x+99
13        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;
14        y = -b+c-e-f+592
15        ys.append(y)
16    return ys
```

Sum-Last7

```
1
2 def f(s):
3     a = 297;b = 198;c = 0;d = 99;e = 0;f = -15;g = 0;
4     ys = []
5     for i in range(20):
6         x = s[i]
7         next_a = -a+d-f+g+480
8         next_b = a-d
9         next_c = d+e-99
10        next_d = -c+99
11        next_e = -b+198
12        next_f = -c+f+x
13        next_g = x
14        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;g = next_g;
15        y = -d+f+114
16        ys.append(y)
17    return ys
```

Current-Number

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

Prev1

```
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+99
8         next_b = -x+99
9         a = next_a;b = next_b;
10        y = a
11        ys.append(y)
12    return ys
```

Prev2

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

Prev3

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

Prev4

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

Prev5

```
1
2 def f(s):
3     a = 0;b = 0;c = 99;d = 99;e = 99;f = 99;
4     ys = []
5     for i in range(20):
6         x = s[i]
7         next_a = -c+99
8         next_b = -d+99
9         next_c = -b+99
10        next_d = e
11        next_e = f
12        next_f = -x+99
13        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;
14        y = a
15        ys.append(y)
16    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
```

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-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
```

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

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

Bit-Dot-Prod-Mod2

```
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 = (not a and b and c) or (a and not b and not c) or (a and not b and c)
8         or (a and b and not c)
9         a = next_a;
10        y = a
11        ys.append(y)
12    return ys
```

Add-Mod-3

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

Newton-Freebody

```
1
2 def f(s):
3     a = 82;b = 393;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = a-x
8         next_b = -a+b+82
9         a = next_a;b = next_b;
10        y = -a+b-311
11        ys.append(y)
12    return ys
```

Newton-Gravity

```
1
2 def f(s):
3     a = 72;b = 513;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = a-x+1
8         next_b = -a+b+x+71
9         a = next_a;b = next_b;
10        y = b-513
11        ys.append(y)
12    return ys
```

Newton-Spring

```
1
2 def f(s):
3     a = 64;b = 57;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = a+b-x-57
8         next_b = -a+121
9         a = next_a;b = next_b;
10        y = -a+64
11        ys.append(y)
12    return ys
```

7.6.10 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

```

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

```

```

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 7.2: 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	4	2	-	1	0	0
2	Base_3_Addition	2	5	2	-	1	0	0
3	Base_4_Addition	2	5	2	-	1	0	0
4	Base_5_Addition	2	5	2	-	1	0	0
5	Base_6_Addition	2	6	2	-	1	2.45e-09	2.53e-09
6	Base_7_Addition	2	10	2	-	1	2.32e-06	2.31e-06
7	Bitwise_Xor	1	2	2	-	1	0	0
8	Bitwise_Or	1	-	1	-	1	3.03e-02	3.03e-02
9	Bitwise_And	1	-	1	-	1	3.03e-02	3.03e-02
10	Bitwise_Not	1	-	1	-	1	0	0
11	Parity_Last2	1	229	2	-	1	1.68e-02	1.69e-02
12	Parity_Last3	2	5	2	-	1	1.62e-04	1.64e-04
13	Parity_Last4	3	29	2	-	1	3.07e-07	2.99e-07
14	Parity_All	1	2	2	-	1	0	0
15	Parity_Zeros	1	2	2	-	1	0	0
16	Evens_Counter	4	73	3	-	1	8.89e-05	8.88e-05
17	Sum_All	1	-	1	-	1	6.09e-08	6.13e-08
18	Sum_Last2	2	-	1	-	1	0	0
19	Sum_Last3	3	-	1	-	1	6.34e-07	6.35e-07
20	Sum_Last4	4	-	1	-	1	2.10e-04	2.11e-04
21	Sum_Last5	5	-	1	-	1	8.86e-03	8.87e-03
22	Sum_Last6	6	-	1	-	1	1.82e-02	1.81e-02
23	Sum_Last7	7	-	1	-	1	3.03e-02	3.01e-02
24	Current_Number	1	-	1	-	1	0	0
25	Prev1	2	-	1	-	1	0	0
26	Prev2	3	-	1	-	1	0	0
27	Prev3	4	-	1	-	1	0	0
28	Prev4	5	-	1	-	1	2.04e-07	2.05e-07
29	Prev5	6	-	1	-	1	6.00e-05	5.96e-05
30	Previous_Equals_Current	2	5	2	-	1	6.72e-05	6.61e-05
31	Diff_Last2	2	-	1	-	1	0	0
32	Abs_Diff	2	-	1	2	2	1.84e-07	1.84e-07
33	Abs_Current	1	2	2	-	1	4.51e-08	5.71e-08
34	Diff_Abs_Values	2	4	2	-	1	3.15e-06	2.96e-06
35	Min_Seen	1	2	2	-	1	0	0
36	Max_Seen	1	2	2	-	1	1.46e-12	0
37	Majority_0_1	1	63	2	-	1	4.03e-03	4.05e-03
38	Majority_0_2	4	98	2	-	1	1.64e-04	1.71e-04
39	Majority_0_3	21	132	3	-	1	6.94e-05	6.86e-05
40	Evens_Detector	5	163	2	-	1	8.18e-04	8.32e-04
41	Perfect_Square_Detector	48	100	2	-	1	1.92e-03	1.97e-03
42	Bit_Palindrome	18	86	2	-	1	3.81e-05	3.69e-05
43	Balanced_Parenthesis	1	16	2	-	1	7.44e-03	7.10e-03
44	Parity_Bits_Mod2	1	-	1	-	1	0	0
45	Alternating_Last3	2	3	2	-	1	1.85e-02	1.87e-02
46	Alternating_Last4	2	3	2	-	1	8.24e-06	8.09e-06
47	Bit_Shift_Right	2	-	1	-	1	0	0
48	Bit_Dot_Prod_Mod2	1	3	2	-	1	0	0
49	Div_3	2	59	2	-	1	6.40e-03	6.43e-03
50	Div_5	4	76	2	-	1	1.50e-04	1.55e-04
51	Div_7	4	103	2	-	1	6.65e-04	6.63e-04
52	Add_Mod_3	1	149	2	-	1	1.02e-03	1.04e-03
53	Add_Mod_4	2	33	2	-	1	1.53e-04	1.44e-04
54	Add_Mod_5	3	43	2	-	1	1.02e-03	1.03e-03
55	Add_Mod_6	4	108	2	-	1	6.14e-04	6.12e-04
56	Add_Mod_7	4	199	2	-	1	3.96e-04	4.07e-04
57	Add_Mod_8	67	134	2	-	1	8.53e-04	8.34e-04
58	Dithering	81	166	2	-	1	7.72e-04	7.75e-04
59	Newton_Freebody	2	-	1	-	1	2.61e-07	2.62e-07
60	Newton_Gravity	2	-	1	-	1	1.81e-07	1.87e-07
61	Newton_Spring	2	-	1	-	1	0	0
62	Newton_Magnetic	4	-	1	-	1	8.59e-05	8.60e-05

Chapter 8

The Geometry of Concepts: Sparse Autoencoder Feature Structure^{*}

This is the first chapter in this thesis on sparse autoencoders (SAEs), which attempt to decompose neural network activation vectors into a sum of sparsely occurring “features”. This work finds a variety of structure to these features. In my section (8.4), on “brain”-lobes, I find that features which point in similar directions also tend to fire together within the same document. This suggests a that a kind of “coarse graining” of SAE features is possible—groups of SAE features, and not just individual SAE features, may reflect meaningful units of the model, raising interesting questions about whether there is any “right” level of analysis for decomposing a neural network.

Abstract

Sparse autoencoders have recently produced dictionaries of high-dimensional vectors corresponding to the universe of concepts represented by large language models. We find that this concept universe has interesting structure at three levels: 1) The “atomic” small-scale structure contains “crystals” whose faces are parallelograms or trapezoids, generalizing well-known examples such as (*man:woman::king:queen*). We find that the quality of such parallelograms and associated function vectors improves greatly when projecting out global distractor directions such as word length, which is efficiently done with linear discriminant analysis. 2) The “brain” intermediate-scale structure has significant spatial modularity; for example, math and code features form a “lobe” akin to functional lobes seen in neural fMRI images. We quantify the spatial locality of these lobes with multiple metrics and find that clusters of co-occurring features, at coarse enough scale, also cluster together spatially far more than one would expect if feature geometry were random. 3) The “galaxy” scale large-scale structure of the feature point cloud is not isotropic, but instead has a power law of eigenvalues with steepest slope in middle layers. We also quantify how the clustering entropy depends on the layer.

^{*}Yuxiao Li, Eric J. Michaud, David D. Baek, Joshua Engels, Xiaoqing Sun, and Max Tegmark. “The geometry of concepts: Sparse autoencoder feature structure.” *Entropy* 27, no. 4 (2025): 344.

8.1 Introduction

While large language models (LLMs) now exhibit a variety of impressive abilities [1–3], we largely do not understand the internal cognition that underlies the behavior of these systems. This lack of transparency may pose a challenge for a variety of AI safety [4] concerns. For instance, it may be difficult to tell whether seemingly-benign model behavior in any particular instance is sycophantic [5] or deceptive [6] without an analysis of the internals of the system. Such “interpretability” analysis has already shown promise in auditing AI systems [7] to identify misaligned goals [8]. As systems become more powerful, there is a need for methods to further our understanding of the internal representations and algorithms learned by these systems [9,10].

The past year has seen a breakthrough in understanding how large language models work: sparse autoencoders (SAEs) have discovered large numbers of vectors (“features”) in their activation space that can be interpreted as concepts [11–13]. These advances build on earlier studies applying sparse coding to artificial neural network representations [14–16], and to earlier work in neuroscience on biological neural representations [17,18]. Underlying this work is the idea that neural networks use *sparse coding* to represent concepts in their activation space [19]. In particular, sparse autoencoders are motivated by the assumptions that (1) networks compute a variety of “features” from their input, (2) features are represented as one-dimensional directions in activation space $\{\mathbf{d}_i\}$, (3) features are represented simply by adding them to the network’s activations, so activation vectors take the form $\sum_i f_i \mathbf{d}_i$, and (4) the coefficients f_i are *sparse* – only a small subset of all possible features “fire” at once. The combination of assumptions (2) - (4) has been called the Linear Representation Hypothesis [20–22].

If these assumptions hold, we could automatically discover these features with *sparse dictionary learning*. Sparse dictionary learning attempts to learn an overcomplete basis (dictionary) $\{\mathbf{d}_i\}$ such that vectors $\mathbf{x}$ from a given distribution can be represented as sparse linear combinations of dictionary elements. Sparse autoencoders offer a simple approach to sparse dictionary learning. Sparse autoencoders consist of a learnable encoder function Enc , which maps vectors $\mathbf{x} \in \mathbb{R}^n$ to a hidden latent representation $\mathbf{f} \in \mathbb{R}^m$, and a decoder Dec which maps latent $\mathbf{f}$ back to $\hat{\mathbf{x}} \in \mathbb{R}^n$. The objective of the sparse autoencoder is to accurately reconstruct the input $\mathbf{x}$ from a **sparse** latent representation, and are trained with gradient descent with a loss function like:

$$\mathcal{L} = \|\mathbf{x} - \text{Dec}(\text{Enc}(\mathbf{x}))\|_2^2 + \lambda \|\mathbf{f}\|_0.$$

Sparse autoencoders use a linear decoder $\text{Dec}(\mathbf{f}) = \mathbf{W}_d \mathbf{f} + \mathbf{b}_d$, so that the output of the SAE can be interpreted as a linear combination of features: $\hat{\mathbf{x}} = \sum_i f_i \mathbf{W}_d^i + \mathbf{b}_d$. In practice, hidden latents discovered by sparse autoencoders tend to be more interpretable than neurons, activating in more consistent contexts [11,12], suggesting that they may be learning the true latents underlying the network’s computation. For AI safety, sparse autoencoders have shown some preliminary success: Ref [7] report specially training an LLM to have a hidden objective, and then challenging separate teams of researchers to identify this objective. One team was able to quickly identify this objective by looking at sparse autoencoder features which activated when the LLM was prompted to exhibit “potentially concerning behaviors”, and then looking at examples in the training data where that same feature fired.

Although some early work motivating sparse autoencoders suggested that networks would arrange features maximally spread apart (approximately orthogonal) [19], recent works have

suggested that features may have a more sophisticated geometric structure [13,22]. Recently, a large collection of SAEs have been made publicly available [23], so it is timely to study their structure at various scales. Thus, the present paper examines sparse autoencoder feature structure at three separate spatial scales, which we refer to informally as the “atom” scale, “brain” scale, and “galaxy” scale. These playful analogies are not meant to be precise, but instead gesture at certain concepts and methods of analysis from other fields which we apply to understanding language model feature structure.

This paper is organized as follows. In Section 8.2, we summarize related work. In Section 8.3, we investigate if the “atomic” small-scale structure contains “crystals” whose faces are parallelograms or trapezoids, generalizing well-known examples such as (*man:woman::king:queen*). In Section 8.4, we test if the “brain” intermediate-scale structure has functional modularity akin to biological brains. In Section 8.5, we study the “galaxy” scale large-scale structure of the feature point cloud, testing whether it is more interestingly shaped and clustered than an isotropic Gaussian distribution, and conclude in Section 8.6.

8.2 Related work

Neural network geometry Many past works have studied the geometry of neural network activations. These works find that the intrinsic dimension of neural network hidden states are much lower than the full model dimension [24], that nearby vectors in activation space are semantically similar [25], and that at local minima well generalizing neural network loss landscapes have many “flat” directions [26]. Other works study how representations evolve through models; one hypothesis is “iterative inference”, which claims that neural networks iteratively refine activations layer by layer [27,28]. A contrasting hypothesis is a circuits view, which holds that information flows in discrete steps along a directed acyclic graph through the model, and representations cleanly change between steps [29]. Another work [30] found that representations of hierarchically related concepts are orthogonal to each other while categorical concepts are represented as polytopes. Our work is in the same vein as these earlier analysis, but differs in an important way because we use the SAE basis, which represents the model’s atomic concept space instead of its activation space.

SAE feature structure: Sparse autoencoders (SAEs) are a recent approach for discovering interpretable language model features without supervision, although relatively few works have examined SAE feature structure. Bricken et al. [12] and Templeton et al. [13] both visualize SAE features with UMAP projections and notice that features tend to group together in “neighborhoods” of related features, in contrast to the approximately-orthogonal geometry observed in the toy model of Elhage et al. [19]. Engels et al. [22] find examples of SAE structure where multiple SAE features appear to reconstruct a multi-dimensional feature with interesting geometry, and multiple authors have recently speculated that SAE vectors might contain more important structures [31,32]. Bussmann et al. [33] suggest that SAE features are in fact linear combinations of more atomic features, and discover these more atomic latents with “meta SAEs”. Our discussion of crystal structure in SAE features is related to this idea that seemingly atomic representations might be composed of more atomic representations.

Semantically Meaningful Linear Representations: Early work found that word embedding methods such as GloVe and Word2vec contained directions encoding semantic concepts, e.g.

the well-known formula $f(\text{king}) - f(\text{man}) + f(\text{woman}) = f(\text{queen})$ [34–36]. More recent research has found similar evidence of linear representations in sequence models trained only on next token prediction, including Othello board positions [37,38], integer lattices [39], the truth value of assertions [40], and numeric quantities such as longitude, latitude, birth year, and death year [41,42], inspiring the Linear Representation Hypothesis (see above). Recent works have also found causal *function vectors* for in-context learning [43–45]. These function vectors induce the model to perform a certain task when added into the model’s hidden states. Our discussion of crystal structures builds upon these previous works by finding these task vectors and parallelogram structures in sparse autoencoder representations.

8.3 “Atom” scale: crystal structure

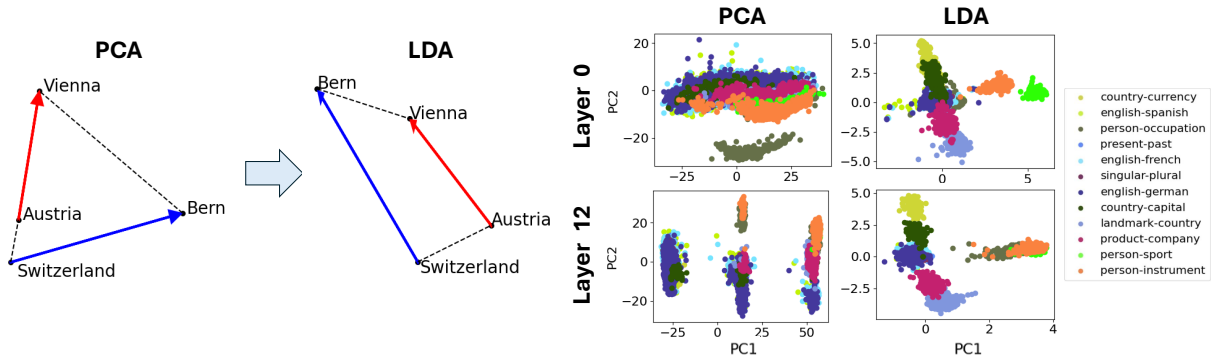


Figure 8.1: Parallelogram and trapezoid structure is revealed (left) when distractor dimensions were projected out from the activations using LDA. LDA results in tighter clusters of pairwise gamma-2-2b activation differences (right), where each cluster corresponds to different semantic transformation. Distractor features are defined as those that are not related to semantics of the text; for instance, the first principal component of gamma-2-2b’s layer 0 activations (top-left figure on the right panel) represents word length. Parallelogram or trapezoid structures suggest that there is a unique direction in the activation space that represents each semantic transformation.

In this section, we search for what we term *crystal* structure in the point cloud of SAE features. By this we mean geometric structure reflecting semantic relations between concepts, generalizing the classic example of $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) = (\text{man}, \text{woman}, \text{king}, \text{queen})$ forming an approximate *parallelogram* where $\mathbf{b} - \mathbf{a} \approx \mathbf{d} - \mathbf{c}$. This can be interpreted in terms of two *function vectors* $\mathbf{b} - \mathbf{a}$ and $\mathbf{c} - \mathbf{a}$ that turn male entities female and turn entities royal, respectively. We also search for *trapezoids* with only one pair of parallel edges $\mathbf{b} - \mathbf{a} \propto \mathbf{d} - \mathbf{c}$ (corresponding to only one function vector); Figure 8.1 (right) shows such an example with $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) = (\text{Austria}, \text{Vienna}, \text{Switzerland}, \text{Bern})$, where the function vector can be interpreted as mapping countries to their capitals. Studying these crystal structures is important because they provide insight into how LLMs internally represent semantic operations and relational knowledge. For instance, function vectors help us assess the extent to which semantic structures within models align with human intuition and language logic, as explored in recent literature [43,46].

We search for crystals by computing all pairwise difference vectors and clustering them using the K-means algorithm [47], where the vectors could be either the original model’s hidden state activations (*model crystal*) or SAE features’ decoder vectors (*SAE crystal*). We use Gemma-2-2b for the experiment. If there is a direction that represents each semantic transformation, we expect each resulting cluster to correspond to each function vector. In other words, any pair of difference vectors in a cluster will form a trapezoid or parallelogram, depending on whether the difference vectors are normalized or not before clustering (or, equivalently, whether we quantify similarity between two difference vectors via Euclidean distance or cosine similarity).

Our initial search for SAE crystals found mostly noise. To investigate why, we decided to focus on activations of the model’s early layer, where many SAE features correspond to a single token. Since SAE feature vectors in the early layers are often closely related to the corresponding model activations, we believed that studying the activations of these early layers could help clarify why our initial crystal search primarily found noise. Therefore, we studied gemma-2-2b residual stream activations for previously reported word $\mapsto$ word function vectors from the dataset of [43]. Figure 8.1 illustrates that candidate crystal quadruplets are typically far from being parallelograms or trapezoids. This is consistent with multiple papers pointing out that (man,woman,king,queen) is not an accurate parallelogram either.

We believe the reason to be the presence of what we term *distractor features*. We define distractor features to be the features that are not related to semantics of the text. For example, we find that the horizontal axis in Figure 8.1 (right) corresponds mainly to word length (Appendix 8.7.2, Figure 8.10), which is semantically irrelevant and wreaks havoc on the trapezoid (left), since “Switzerland” is much longer than the other words. However, these distractor features were not always interpretable; in some cases, it was difficult to associate features with any clear linguistic property.

To eliminate such semantically irrelevant distractor vectors, we wish to project the data onto a lower-dimensional subspace orthogonal to them. For the [43] dataset, we do this with Linear Discriminant Analysis (LDA) [48], which projects onto signal-to-noise eigenmodes where “signal” and “noise” are defined as the covariance matrices of inter-cluster variation and intra-cluster variation, respectively. Figure 8.1 illustrates that this dramatically improves the cluster and trapezoid/parallelogram quality, highlighting that distractor features can hide existing crystals.

8.4 “Brain” scale: Meso-scale modular structure

We now zoom out and look for larger-scale structure. In particular, we investigate if *functionally* similar groups of SAE features (which tend to fire together) are also *geometrically* similar, forming “lobes” in the activation space. We refer to this analysis as “brain” scale because, in animal brains, functionally similar groups of neurons also typically cluster together spatially. For example, Broca’s area is involved in speech production, the auditory cortex processes sound, and the amygdala is primarily associated with processing emotions. We are curious whether we can find analogous functional modularity in the SAE feature space. While prior work has qualitatively observed that semantically related features are spatially close via UMAP projections of features [12,13], we aim to more precisely quantify the relationship between functional similarity and spatial similarity.

We test a variety of methods for automatically discovering such functional “lobes” and for quantifying if they are spatially modular. We define a **lobe partition** as a partition of the SAE

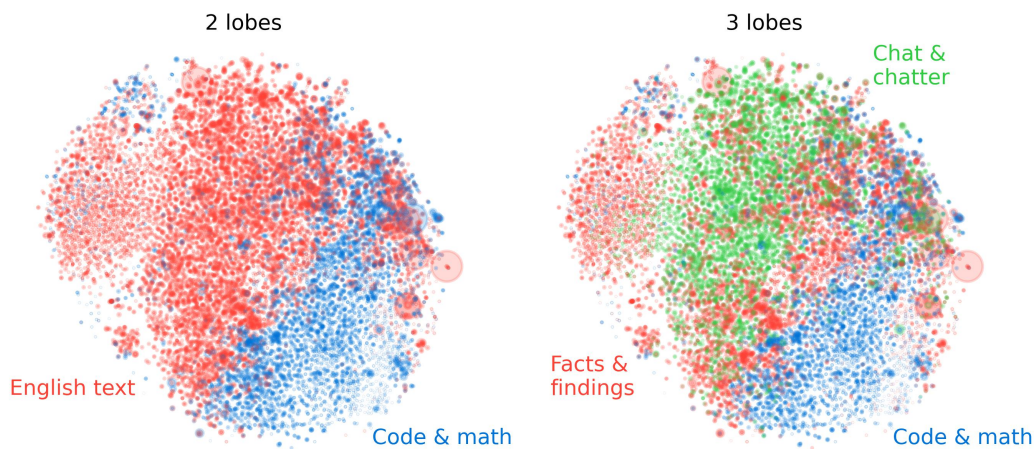


Figure 8.2: Features in the SAE feature point cloud identified that tend to fire together within documents are seen to also be geometrically co-located in functional “lobes”, here down-projected to 2D with t-SNE with point size proportional to feature frequency. A 2-lobe partition (left) is seen to break the point cloud into roughly equal parts, active on code/math documents and English language documents, respectively. A 3-lobe partition (right) is seen to mainly subdivide the English lobe into a part for short messages and dialogue (e.g. chat rooms and parliament proceedings) and one primarily containing long-form scientific papers.

feature point cloud into k subsets (“lobes”) that are computed *without positional information*. Instead, we identify such lobes based on them being *functionally* related, specifically, tending to fire together within a document.

To automatically identify functional lobes, we first compute a histogram of SAE feature co-occurrences. We take gemma-2-2b and pass documents from The Pile [49] through it. In this section, we report results with a layer 12 residual stream SAE with 16k features and average L0 of 41. For this SAE, we record the features that fire (we count a feature i as firing if its encoder assigns it a coefficient $f_i > 1$). Features are counted as co-occurring if they both fire within the same block of 256 tokens – this length provides a coarse “time resolution” allowing us to find tokens that tend to fire together within the same document rather than just at the same token. We use a max context length of 1024, and only use one such context per document, giving us at most 4 blocks (and histogram updates) per document of The Pile. We compute histograms across 50k documents. Given this histogram, we compute an affinity score between each pair of SAE features based on their co-occurrence statistics and perform spectral clustering on the resulting affinity matrix. We use the spectral clustering implementation of scikit-learn [50] with default settings with varying choice of `n_clusters`.

In Figure 8.2, we visualize lobes discovered with this method with `n_clusters=2,3` via a t-SNE projection [51]. For this figure, we used the “phi coefficient” as the measure of co-occurrence similarity between features. We find that lobes visually appear to be spatially localized. For instance, features which fire primarily on math and code documents tend to cluster together spatially.

We experiment with the following notions of co-occurrence-based affinity: simple matching coefficient, Jaccard similarity, Dice coefficient, overlap coefficient, and Phi coefficient, which can

all be computed just from a co-occurrence histogram. In the Appendix 8.7.1, we review definitions for each of these and in Figure 8.3 illustrate how the choice between them affects the resulting lobe t-SNE plots. We also show how lobes appear when we cluster based on geometry directly using cosine similarities, as described below.

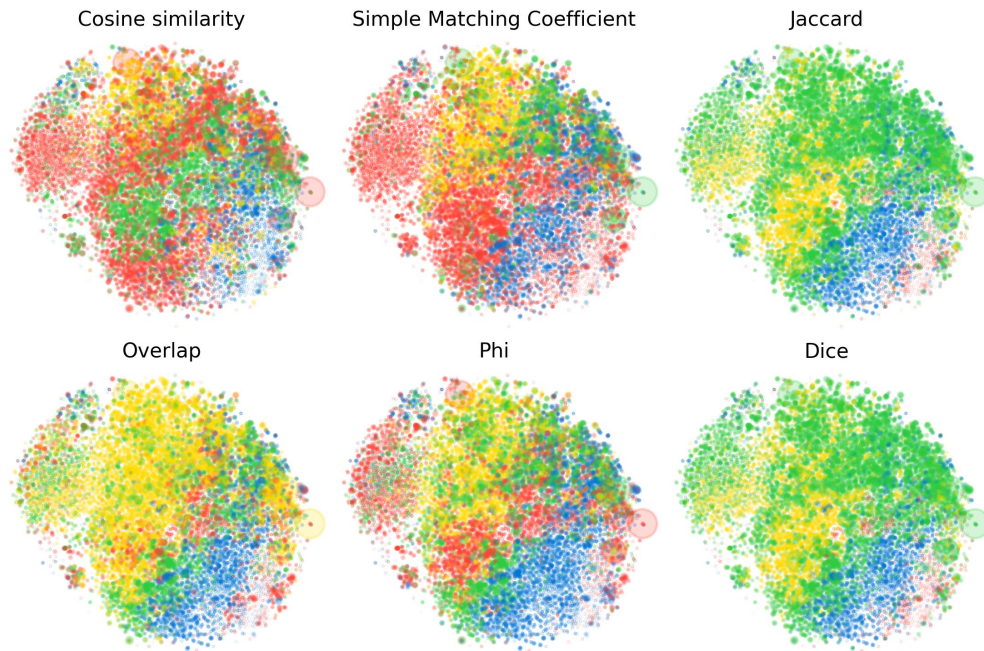


Figure 8.3: Comparison of the lobe partitions of the SAE point cloud discovered with different affinity measures, with the same t-SNE projection as Figure 8.2. In the top left, we show clusters computed from **geometry**, the cosine similarity between features as the affinity score for spectral clustering. All other measures are based on whether SAE features co-occur (fire together) within 256-token blocks, using different measures of affinity. Although the Phi coefficient predicts spatial structure best, all co-occurrence measures are seen to discover the “code/math lobe”.

While these plots show a qualitative relationship between co-occurrence and feature geometry, we aim to quantify this relationship. Our null hypothesis is that functionally similar points (of commonly co-occurring SAE features) are uniformly distributed throughout the activation space, showing no spatial modularity. To quantify how statistically significant this is, we use two approaches to rule out the null hypothesis:

1. While we can cluster features based on whether they co-occur, we can also perform spectral clustering based on the cosine similarity between SAE feature decoder vectors. So instead of feature affinity values being e.g. their co-occurrence phi coefficient, affinity matrix values are instead computed simply from feature geometry as $A_{ij} = \mathbf{d}_i \cdot \mathbf{d}_j$. Given a clustering of SAE features using cosine similarity and a clustering using co-occurrence, we compute the mutual information between these two sets of labels. In some sense, this measures the amount of information about geometric structure that one gets from knowing functional structure. We report the adjusted mutual information [52] as implemented by scikit-learn [50], which corrects for chance agreements between the clusters.

2. Another conceptually simple approach is to train models to predict which functional lobe a feature is in from its geometry. To do this, we take a given set of lobe labels from our co-occurrence-based clustering, and train a logistic regression model to predict these labels directly from the point positions, using an 80-20 train-test split and reporting the balanced test accuracy of this classifier.

Figure 8.4 shows that for both measures, the Phi coefficient gives the best correspondence between functional lobes and feature geometry. To show that this is statistically significant, we randomly permute the cluster labels from the cosine similarity-based clustering and measure the adjusted mutual information. We also re-initialize the SAE feature decoder directions from a random Gaussian and normalize, and then train logistic regression models to predict functional lobe from these random feature directions. Figure 8.4 (bottom) shows that both tests rule out the null hypothesis with high significance, at 954 and 74 standard deviations, respectively, clearly demonstrating that the lobes we see are real and not a statistical fluke.

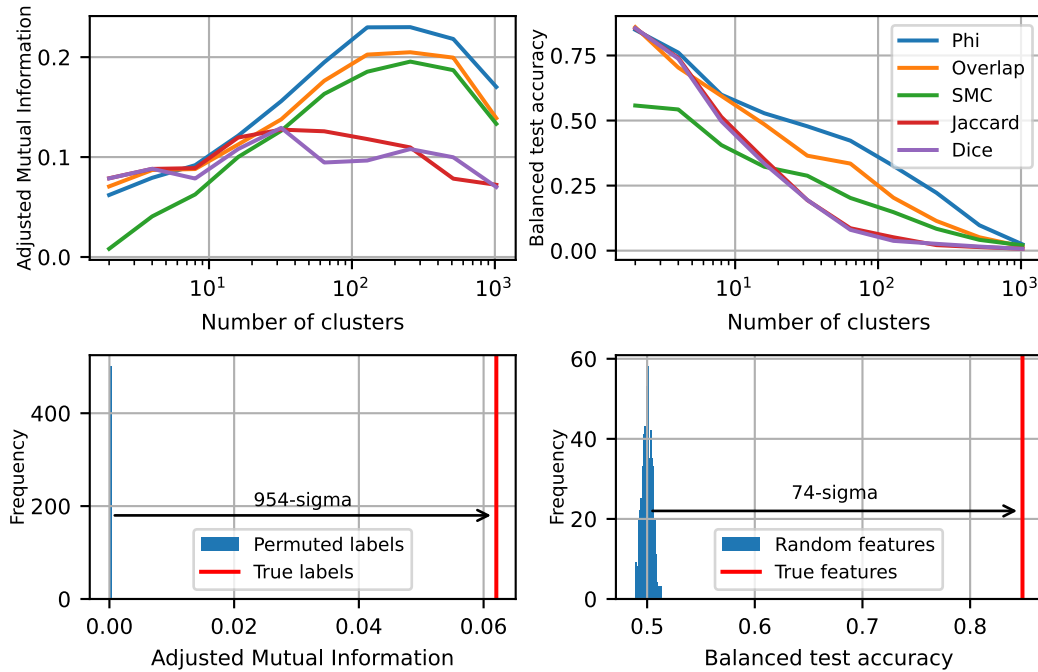


Figure 8.4: Top Left: Adjusted mutual information between spatial clusters and functional (cooccurrence-based) clusters. Top Right: logistic regression balanced test accuracy, predicting cooccurrence-based cluster label from position. Bottom Left: Adjusted mutual information with randomly-permuted cosine similarity-based clustering labels. Bottom Right: balanced test accuracy with random unit-norm feature vectors. The statistical significance reported is for Phi-based clustering into lobes.

To assess what each lobe specializes in, we run 10k documents from The Pile through gemma-2-2b, and again record which SAE features at layer 12 fire within blocks of 256 tokens. For each block of tokens, we record which lobe has the highest proportion of its features firing. Each document in The Pile is attached with a name specifying the subset of the corpus that document is from. For each document type, for each 256-token block within a document of that type, we

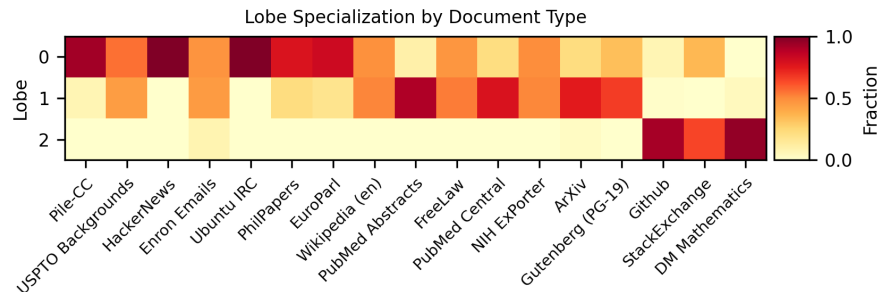


Figure 8.5: Fraction of contexts in which each lobe had the highest proportion of activating features. For each document type, these fractions sum to 1 across the lobes. We see that lobe 2 typically disproportionately activates on code and math documents. Lobe 0 and 1 activate on other documents, with lobe 0 activating more on documents containing short text and dialogue (chat comments, parliamentary proceedings) and lobe 1 activating more on scientific papers.

record which lobe which had the highest proportion of its SAE features firing. Across thousands of documents, we can then look at a histogram of which lobes were maximally activating across each document type. We show these results for three lobes, computed with the Phi coefficient as the co-occurrence measure, in Figure 8.5. This forms the basis for our lobe labeling in Figure 8.2.

These findings raise interesting questions about whether individual sparse autoencoder features are the most natural units for understanding neural networks [53,54]. In biological brains, one can study individual neurons, groups of neurons, groups of groups of neurons, and so on up to very large-scale structures, and it is not clear a priori what “scale” of analysis will be most fruitful [55]. We may face a similar ambiguity with sparse autoencoder features, since, as we have seen, groups of co-occurring, geometrically-related features can be interpretable and studied in their own right. This question, of whether there is a right “scale” of analysis for SAE features, is made even more salient by the observation in prior work of “feature splitting” [12].

8.5 “Galaxy” scale: Large-scale point cloud structure

In this section, we further broaden our perspective to analyze the “galaxy” scale structure of the point cloud, focusing on its overall shape and clustering properties. This analysis is loosely inspired by work in astronomy [56] characterizing the shape [57] and substructure [58] of galaxies. We start by formulating a simple null hypothesis: *The point cloud is drawn from an isotropic multivariate Gaussian distribution.*

To test this, we analyze the covariance of the data. As illustrated in Figure 8.6, the eigenvalue spectrum deviates from isotropy, meaning the cloud exhibits directional structure rather than being purely spherical. Even within the first three principal components, the point cloud is anisotropic, with some principal axes slightly wider than others.

To quantify these deviations, we analyze the eigenvalue spectrum of the covariance matrix, comparing it to theoretical expectations from random matrix theory (RMT).

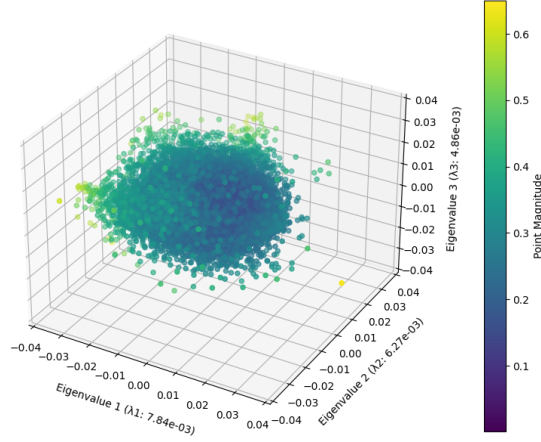


Figure 8.6: 3D Point Cloud visualizations of top PCA components for the gemma2-2b layer 12 SAE features.

8.5.1 Shape analysis

In RMT, the covariance matrix of N random vectors from a multivariate Gaussian distribution follow a Wishart Distribution [59]. Under this assumption, we would expect the eigenvalues to be relatively uniform or to follow the Marcenko-Pastur law [60]. In contrast, we observe a surprising derivation:

- The eigenvalue spectrum of the point cloud decays as a power law rather than following the expected Wishart behavior.
- As shown in Figure 8.6, this power law decay is more pronounced in SAE features compared to raw activations.

Since the abrupt drop off seen for the smallest eigenvalues is caused by limited data and vanishes in the limit $N \rightarrow \infty$, we dimensionally reduce the point cloud to its 100 largest principal components for all subsequent analysis in this section. We describe the shape of this high-dimensional point cloud as resembling a "fractal cucumber", whose width in successive dimensions falls off like a power law. We find such power law scaling is significantly less prominent for activations than for SAE features; it will be interesting for further work to investigate its origins.

Figure 8.7 (left) shows how the slope of the aforementioned power law depends on LLM layer, computed via linear regression against the 100 largest eigenvalues. We see a clear pattern where middle layers have the steepest power law slopes: (Layer 12 has slope -0.47, while early and late layers (e.g., Layers 0 and 24) have shallower slopes (-0.24 and -0.25), respectively). This may hint that middle layers act as a bottleneck, compressing information into fewer principal components, perhaps optimizing for more efficient representation of high-level abstractions. Figure 8.7 (right) compares the eigenvalue spectra of SAE features and neural activations, indicating a significantly steeper power law decay for SAE features. Activations, in contrast, exhibit a much slower decay, indicating weaker power-law behavior and distinct geometric structures in the latent space. Figure 8.8 (left) explores the effective cloud volume (the determinant of the covariance matrix) of the point cloud, quantified by the log-determinant of the covariance matrix across layer. This volume

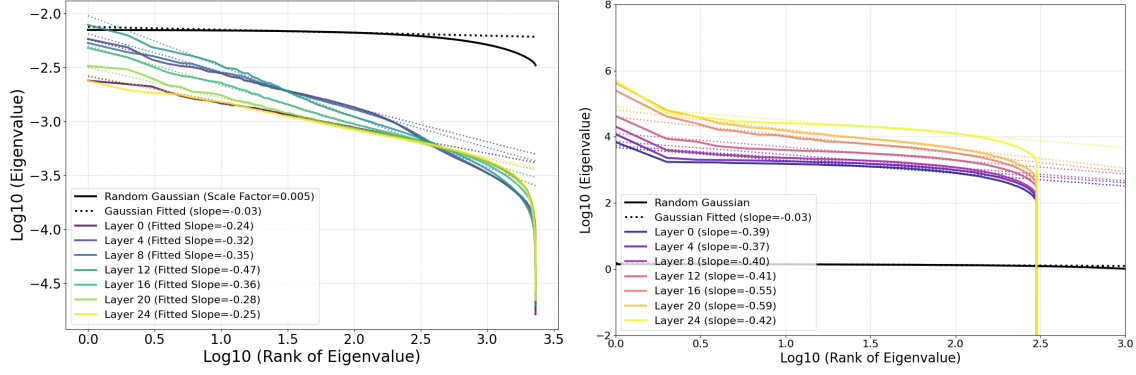


Figure 8.7: Eigenvalue distributions for SAE features and activations. Eigenvalues of the covariance matrix for SAE features (left) and neural activations (right) decay approximately as a power law, with slopes varying across layers. A scaled isotropic Gaussian spectrum is shown for comparison, highlighting the significantly steeper decay for SAE features. Eigenvalue spectra for activations show a much slower decay compared to SAE features, indicating weaker power-law behavior and distinct geometric structures.

variation further reflects the layer-specific changes in the structure and complexity of the latent space.

8.5.2 Clustering analysis

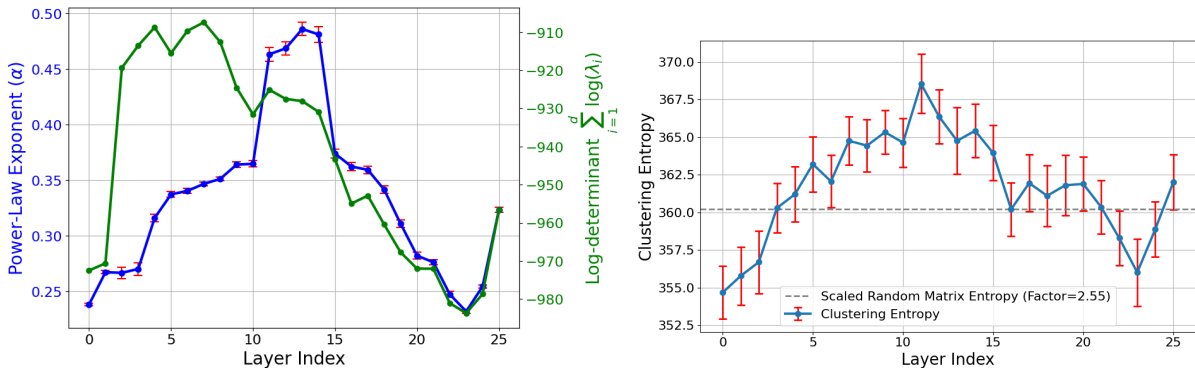


Figure 8.8: Layer-Wise Analysis of Latent Representations. Left: The power-law slope (α) of the eigenvalue spectrum (blue) and the log-determinant of the covariance matrix (green) vary across layers. Both metrics peak in intermediate layers, indicating significant structural changes in the latent space. Right: Estimated clustering entropy across layers with 95% confidence intervals. Middle layers exhibit reduced clustering entropy, while earlier and later layers show higher entropy, reflecting distributed and concentrated feature representations, respectively.

Clustering of galaxies or microscopic particles is often quantified in terms of a power spectrum or correlation function. This is complicated for our very high-dimensional data, since the underlying density varies with radius and, for a high-dimensional Gaussian distribution, is strongly concentrated around a relatively thin spherical shell. For this reason, we instead quantify cluster-

ing by estimating the *entropy* of the distribution that the point cloud is assumed to be sampled from. We estimate the entropy H from our SAE feature point cloud using the k -th nearest neighbor (k-NN) method [61,62], computed as follows,

$$H_{features} = \frac{d}{n} \sum_{i=1}^n \log(r_i + \theta) + \log(n - 1) - \Psi \quad (8.1)$$

where r_i is the distance to the k -th nearest neighbor for point i , and d is the dimensionality of the point cloud; n is the number of points; the constant Ψ is the digamma term from the k-NN estimation. As a baseline, the Gaussian entropy represents the maximum possible entropy for a given covariance matrix. For a Gaussian distribution with the same covariance matrix, the entropy computed as:

$$H_{gauss} = \frac{d}{2} (1 + \log(2\pi)) + \sum_{i=1}^d \log(\lambda_i) \quad (8.2)$$

where λ_i are the eigenvalues of the covariance matrix. We define the **clustering entropy** (often referred to as “negentropy” in physics as $H_{gauss} - H$, i.e., how much lower the entropy is than its maximum allowed value).

The estimated clustering entropy is shown in Figure 8.8 (right), plotted across different layers. The result indicate that the SAE point cloud is strongly clustered, particularly in the middle layers. This observation aligns with the reduced clustering entropy seen at intermediate layers, suggesting significant structural differences in the latent representations.

In future work, it will be interesting to investigate whether these variations depend mainly on the prominence of crystals or lobes in different layers, or have an altogether different origin (entirely different underlying mechanisms).

8.6 Conclusion

We have searched for structure in the SAE concept universe at three levels: 1) The “atomic” small-scale structure contains “crystals” whose faces are parallelograms or trapezoids, generalizing well-known examples such as (*man:woman::king:queen*), may be revealed when projecting out semantically irrelevant distractor features. 2) The “brain” intermediate-scale structure has significant spatial modularity; for example, math and code features form a “lobe” akin to functional lobes seen in neural fMRI images. 3) The “galaxy” scale large-scale structure of the feature point cloud is not isotropic, but instead has a power law of eigenvalues with steepest slope in middle layers.

While we have observed that SAE features exhibit geometric structure at multiple scales, we have not explained *why* this structure forms. We think that further work which not only studies the structure of SAE features, but also seeks to explain the origin of this structure, could be highly valuable. Such work may lead to refinements to our theory of how networks represent features in superposition or to insights which improve sparse autoencoder performance.

We hope that our findings serve as a stepping stone toward deeper understanding of SAE features and the workings of large language models, and that this deeper understanding will eventually help to improve the safety of AI systems as they continue to grow in power.

References

- [1] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al. “Gpt-4o system card.” *arXiv preprint arXiv:2410.21276*, 2024.
- [2] “The Claude 3 Model Family: Opus, Sonnet, Haiku.” In: URL: <https://api.semanticscholar.org/CorpusID:268232499>.
- [3] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.” *arXiv preprint arXiv:2501.12948*, 2025.
- [4] P. Slattery, A. K. Saeri, E. A. Grundy, J. Graham, M. Noetel, R. Uuk, J. Dao, S. Pour, S. Casper, and N. Thompson. “The ai risk repository: A comprehensive meta-review, database, and taxonomy of risks from artificial intelligence.” *arXiv preprint arXiv:2408.12622*, 2024.
- [5] M. Sharma, M. Tong, T. Korbak, D. Duvenaud, A. Askill, S. R. Bowman, N. Cheng, E. Durmus, Z. Hatfield-Dodds, S. R. Johnston, et al. “Towards understanding sycophancy in language models.” *arXiv preprint arXiv:2310.13548*, 2023.
- [6] P. S. Park, S. Goldstein, A. O’Gara, M. Chen, and D. Hendrycks. “AI deception: A survey of examples, risks, and potential solutions. arXiv.” *arXiv preprint arXiv:2308.14752*, 2023.
- [7] S. Marks et al. “Auditing Language Models for Hidden Objectives.” *arXiv preprint*, 2024.
- [8] R. Ngo, L. Chan, and S. Mindermann. “The alignment problem from a deep learning perspective.” *arXiv preprint arXiv:2209.00626*, 2022.
- [9] L. Bereska and E. Gavves. “Mechanistic Interpretability for AI Safety—A Review.” *arXiv preprint arXiv:2404.14082*, 2024.
- [10] L. Sharkey, B. Chughtai, J. Batson, J. Lindsey, J. Wu, L. Bushnaq, N. Goldowsky-Dill, S. Heimersheim, A. Ortega, J. Bloom, et al. “Open Problems in Mechanistic Interpretability.” *arXiv preprint arXiv:2501.16496*, 2025.
- [11] R. Huben, H. Cunningham, L. R. Smith, A. Ewart, and L. Sharkey. “Sparse Autoencoders Find Highly Interpretable Features in Language Models.” In: *The Twelfth International Conference on Learning Representations*. 2023.
- [12] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. URL: <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [13] A. Templeton et al. “Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet.” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html>.
- [14] M. Faruqui, Y. Tsvetkov, D. Yogatama, C. Dyer, and N. Smith. “Sparse overcomplete word vector representations.” *arXiv preprint arXiv:1506.02004*, 2015.
- [15] J. Zhang, Y. Chen, B. Cheung, and B. A. Olshausen. “Word embedding visualization via dictionary learning.” *arXiv preprint arXiv:1910.03833*, 2019.

- [16] Z. Yun, Y. Chen, B. A. Olshausen, and Y. LeCun. “Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors.” *arXiv preprint arXiv:2103.15949*, 2021.
- [17] B. A. Olshausen and D. J. Field. “Emergence of simple-cell receptive field properties by learning a sparse code for natural images.” *Nature*, **381**(6583), 1996, pp. 607–609.
- [18] B. A. Olshausen and D. J. Field. “Sparse coding with an overcomplete basis set: A strategy employed by V1?” *Vision research*, **37**(23), 1997, pp. 3311–3325.
- [19] N. Elhage et al. “Toy Models of Superposition.” *Transformer Circuits Thread*, 2022.
- [20] K. Park, Y. J. Choe, and V. Veitch. “The linear representation hypothesis and the geometry of large language models.” *arXiv preprint arXiv:2311.03658*, 2023.
- [21] C. Olah. “What is a Linear Representation? What is a Multidimensional Feature?” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/july-update/index.html#linear-representations>.
- [22] J. Engels, I. Liao, E. J. Michaud, W. Gurnee, and M. Tegmark. “Not All Language Model Features Are Linear.” *arXiv preprint arXiv:2405.14860*, 2024.
- [23] T. Lieberum, S. Rajamanoharan, A. Conmy, L. Smith, N. Sonnerat, V. Varma, J. Kramár, A. Dragan, R. Shah, and N. Nanda. “Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2.” *arXiv preprint arXiv:2408.05147*, 2024.
- [24] A. Ansuini, A. Laio, J. H. Macke, and D. Zoccolan. “Intrinsic dimension of data representations in deep neural networks.” *Advances in Neural Information Processing Systems*, **32**, 2019.
- [25] D. Chandrasekaran and V. Mago. “Evolution of semantic similarity—a survey.” *ACM Computing Surveys (CSUR)*, **54**(2), 2021, pp. 1–37.
- [26] S. Watanabe. *Algebraic geometry and statistical learning theory*. Vol. 25. Cambridge university press, 2009.
- [27] C. Rushing and N. Nanda. “Explorations of Self-Repair in Language Models.” *arXiv preprint arXiv:2402.15390*, 2024.
- [28] N. Belrose, Z. Furman, L. Smith, D. Halawi, I. Ostrovsky, L. McKinney, S. Biderman, and J. Steinhardt. “Eliciting latent predictions from transformers with the tuned lens.” *arXiv preprint arXiv:2303.08112*, 2023.
- [29] A. Conmy, A. Mavor-Parker, A. Lynch, S. Heimersheim, and A. Garriga-Alonso. “Towards automated circuit discovery for mechanistic interpretability.” *Advances in Neural Information Processing Systems*, **36**, 2023, pp. 16318–16352.
- [30] K. Park, Y. J. Choe, Y. Jiang, and V. Veitch. “The geometry of categorical and hierarchical concepts in large language models.” *arXiv preprint arXiv:2406.01506*, 2024.
- [31] J. Mendel. “SAE feature geometry is outside the superposition hypothesis.” *AI Alignment Forum*, 2024. URL: <https://www.alignmentforum.org/posts/MFBTjb2qf3ziWmzz6/sae-feature-geometry-is-outside-the-superposition-hypothesis>.

- [32] L. Smith. “The ‘strong’ feature hypothesis could be wrong.” *AI Alignment Forum*, 2024. URL: <https://www.alignmentforum.org/posts/tojtPCCRpKLSHBdpn/the-strong-feature-hypothesis-could-be-wrong>.
- [33] B. Bussmann, M. Pearce, P. Leask, J. I. Bloom, L. Sharkey, and N. Nanda. *Showing SAE Latents Are Not Atomic Using Meta-SAEs*. AI Alignment Forum. Accessed on October 1, 2024. Aug. 2024. URL: <https://www.alignmentforum.org/posts/TMAmHh4DdMr4nCSr5/showing-sae-latents-are-not-atomic-using-meta-saes>.
- [34] A. Drozd, A. Gladkova, and S. Matsuoka. “Word embeddings, analogies, and machine learning: Beyond king-man+ woman= queen.” In: *Proceedings of coling 2016, the 26th international conference on computational linguistics: Technical papers*. 2016, pp. 3519–3530.
- [35] J. Pennington, R. Socher, and C. D. Manning. “Glove: Global vectors for word representation.” In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.
- [36] L. Ma and Y. Zhang. “Using Word2Vec to process big text data.” In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 2895–2897.
- [37] N. Nanda, A. Lee, and M. Wattenberg. “Emergent linear representations in world models of self-supervised sequence models.” *arXiv preprint arXiv:2309.00941*, 2023.
- [38] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. “Emergent world representations: Exploring a sequence model trained on a synthetic task.” *arXiv preprint arXiv:2210.13382*, 2022.
- [39] E. J. Michaud, I. Liao, V. Lad, Z. Liu, A. Mudide, C. Loughridge, Z. C. Guo, T. R. Kheirkhah, M. Vukelić, and M. Tegmark. “Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code.” *Entropy*, 26(12), 2024, p. 1046.
- [40] S. Marks and M. Tegmark. “The geometry of truth: Emergent linear structure in large language model representations of true/false datasets.” *arXiv preprint arXiv:2310.06824*, 2023.
- [41] W. Gurnee and M. Tegmark. “Language models represent space and time.” *arXiv preprint arXiv:2310.02207*, 2023.
- [42] B. Heinzerling and K. Inui. “Monotonic representation of numeric properties in language models.” *arXiv preprint arXiv:2403.10381*, 2024.
- [43] E. Todd, M. L. Li, A. S. Sharma, A. Mueller, B. C. Wallace, and D. Bau. “Function vectors in large language models.” *arXiv preprint arXiv:2310.15213*, 2023.
- [44] R. Hendel, M. Geva, and A. Globerson. “In-context learning creates task vectors.” *arXiv preprint arXiv:2310.15916*, 2023.
- [45] D. Kharlapenko, neverix, N. Nanda, and A. Conmy. *Extracting SAE task features for in-context learning*. AI Alignment Forum. Accessed on October 1, 2024. Aug. 2024. URL: <https://www.alignmentforum.org/posts/5FGXmJ3wqgGRcbyH7/extracting-sae-task-features-for-in-context-learning>.
- [46] K. Meng, D. Bau, A. Andonian, and Y. Belinkov. “Locating and editing factual associations in gpt.” *Advances in neural information processing systems*, 35, 2022, pp. 17359–17372.

- [47] M. Ahmed, R. Seraj, and S. M. S. Islam. “The k-means algorithm: A comprehensive survey and performance evaluation.” *Electronics*, 9(8), 2020, p. 1295.
- [48] P. Xanthopoulos, P. M. Pardalos, T. B. Trafalis, P. Xanthopoulos, P. M. Pardalos, and T. B. Trafalis. “Linear discriminant analysis.” *Robust data mining*, 2013, pp. 27–33.
- [49] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. “The pile: An 800gb dataset of diverse text for language modeling.” *arXiv preprint arXiv:2101.00027*, 2020.
- [50] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python.” *Journal of Machine Learning Research*, 12, 2011, pp. 2825–2830.
- [51] L. Van der Maaten and G. Hinton. “Visualizing data using t-SNE.” *Journal of machine learning research*, 9(11), 2008.
- [52] N. X. Vinh, J. Epps, and J. Bailey. “Information theoretic measures for clusterings comparison: is a correction for chance necessary?” In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 1073–1080.
- [53] A. Mueller, J. Brinkmann, M. Li, S. Marks, K. Pal, N. Prakash, C. Rager, A. Sankaranarayanan, A. S. Sharma, J. Sun, et al. “The quest for the right mediator: A history, survey, and theoretical grounding of causal interpretability.” *arXiv preprint arXiv:2408.01416*, 2024.
- [54] C. Olah. *Transformer Circuits Thread: Interpretability Dreams*. An informal note on future goals for mechanistic interpretability. May 2023. URL: <https://transformer-circuits.pub/2023/interpretability-dreams/index.html> (visited on 05/24/2023).
- [55] E. P. Hoel, L. Albantakis, and G. Tononi. “Quantifying causal emergence shows that macro can beat micro.” *Proceedings of the National Academy of Sciences*, 110(49), 2013, pp. 19790–19795.
- [56] R. C. Kennicutt Jr. “Star formation in galaxies along the Hubble sequence.” *Annual Review of Astronomy and Astrophysics*, 36(1), 1998, pp. 189–231.
- [57] E. P. Hubble. “Extragalactic Nebulae.” *Astrophysical Journal*, 64, Dec. 1926, pp. 321–369. DOI: [10.1086/143018](https://doi.org/10.1086/143018).
- [58] A. Kravtsov. “Dark matter substructure and dwarf galactic satellites.” *Advances in Astronomy*, 2010(1), 2010, p. 281913.
- [59] J. Wishart. “The generalised product moment distribution in samples from a normal multi-variate population.” *Biometrika*, 20(1/2), 1928, pp. 32–52.
- [60] V. Marchenko and L. A. Pastur. “Distribution of eigenvalues for some sets of random matrices.” *Mat. Sb.(NS)*, 72(114), 1967, p. 4.
- [61] B. V. Dasarathy. “Nearest neighbor (NN) norms: NN pattern classification techniques.” *IEEE Computer Society Tutorial*, 1991.
- [62] L. F. Kozachenko and N. N. Leonenko. “Sample estimate of the entropy of a random vector.” *Problemy Peredachi Informatsii*, 23(2), 1987, pp. 9–16.
- [63] P. Jaccard. “Nouvelles Recherches Sur La Distribution Florale.” *Bulletin de la Société vaudoise des Sciences Naturelles*, 44, 1908, pp. 223–270.

- [64] L. R. Dice. “Measures of the Amount of Ecologic Association Between Species.” *Ecology*, 26(3), 1945, pp. 297–302.
- [65] G. U. Yule. “On the Methods of Measuring Association Between Two Attributes.” *Journal of the Royal Statistical Society*, 75(6), 1912, pp. 579–652.

8.7 Appendix

8.7.1 Additional Information on Brain Lobes

Co-occurrence measures

Definitions of co-occurrence based affinity measures: Let n_{ij} be the number of times features i and j co-occur. Let m_{11} be number of times i and j co-occur, m_{00} be number of times i and j both do not occur, m_{10} be number of times i occurs but j does not, $m_{1\bullet}$ be number of times i occurs and j either occurs or not, and so on. Then,

Jaccard similarity, [63]:

$$J_{ij} = \frac{|i \cap j|}{|i \cup j|} = \frac{n_{ij}}{n_{ii} + n_{jj} - n_{ij}}$$

Dice score, [64]:

$$DSC_{ij} = \frac{2|i \cap j|}{|i| + |j|} = \frac{2n_{ij}}{n_{ii} + n_{jj}}$$

Overlap coefficient:

$$overlap_{ij} = \frac{|i \cap j|}{\min(|i|, |j|)} = \frac{n_{ij}}{\min(n_{ii}, n_{jj})}$$

Simple matching coefficient:

$$SMC_{ij} = \frac{m_{00} + m_{11}}{m_{00} + m_{11} + m_{01} + m_{10}}$$

Phi coefficient, [65]:

$$\phi_{ij} = \frac{m_{11}m_{00} - m_{10}m_{01}}{\sqrt{m_{1\bullet}m_{0\bullet}m_{\bullet 1}m_{\bullet 0}}}$$

8.7.2 Understanding Principal Components in Difference Space

Figure 8.10 shows that the first principal component encodes mainly the length difference between two words’ last tokens in gemma-2-2b Layer 0.

8.7.3 Breaking Down SAE Vectors by PCA Component

An additional investigation of structure we undertake is quantifying how SAE vectors are distributed throughout the PCA components of the activations vectors. To do this, we define a PCA score:

$$\text{PCA score}(feature_j) = \frac{1}{n} \sum_i i * (pca_i @ feature_j)^2$$

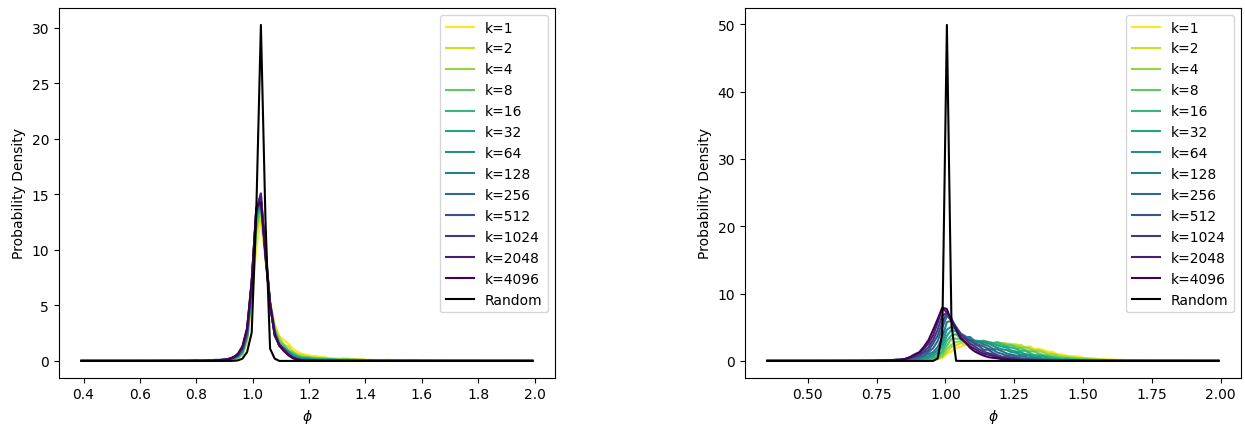


Figure 8.9: Histogram, over all features, of Phi coefficient with k -th nearest cosine similarity neighbor for (left) Layer 0 and (right) Layer 12. Each line represents a different k . The “random” line is plotted by drawing a random feature for each feature, then computing the Phi coefficient. Features with higher cosine similarity have higher Phi coefficient, but this is less pronounced in layer 0 compared to layer 12.

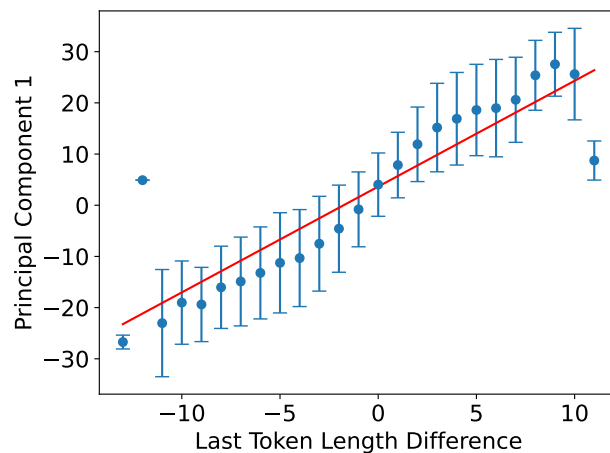


Figure 8.10: Plot of the first principal component in the difference space as a function of last token length difference in gemma-2-2b layer 0. The linear relationship indicates that the first principal component encodes the length difference between two words’ last tokens.

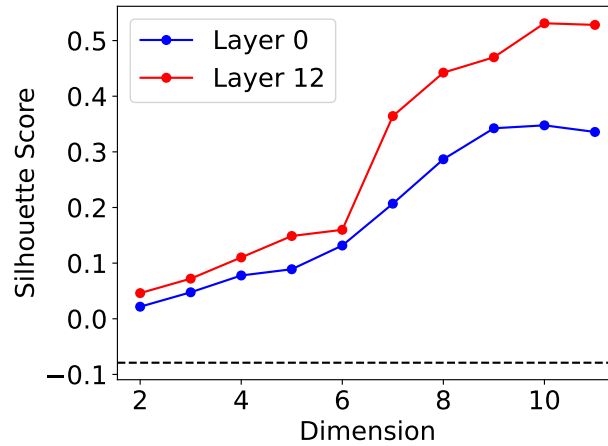


Figure 8.11: Silhouette Score, a measure of clustering quality, as a function of reduced dimension in LDA. The plot indicates that training an affine transformation for semantic cluster separation is easier in mid-layer (layer 12), where the model starts to develop concept-level understanding of the input.

This metric is a weighted sum between 0 and 1 measuring approximately where in the PCA each SAE feature lies. In Figure 8.12, we plot this metric on a single Gemma Scope SAE (the results look similar on all Gemma Scope SAEs), and we see that there is an intriguing dip into earlier PCA features in the last third of SAE features.

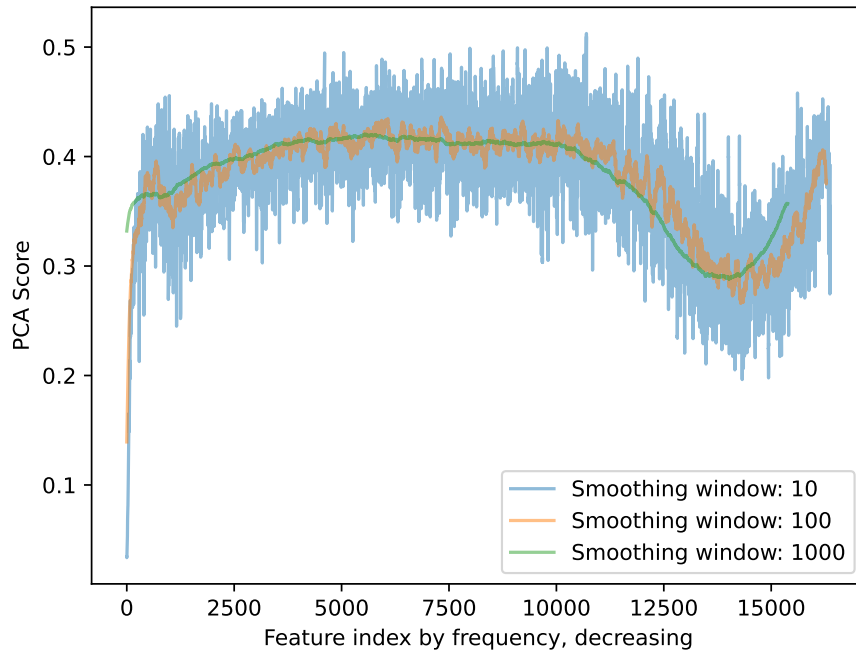


Figure 8.12: Smoothed PCA scores for each SAE feature of the layer 12, width $16k$, $L_0 = 176$ Gemma Scope 2b SAE, sorted by frequency. PCA score = $\frac{1}{n} \sum_i i * (pca_i@feature_j)^2$, where n is the number of PCA features. The smoothed curves just average this somewhat noisy metric over adjacent sorted features. This measures approximately where in the PCA each SAE feature lies, and shows that there is a dip into earlier PCA features in the last third of SAE features.

Chapter 9

Not All Language Model Features Are One-Dimensionally Linear^{*}

This paper finds evidence of multi-dimensional representations (also called feature manifolds) in large language models. When feature manifolds are present, individual SAE features may not be “canonical” units of analysis, since there are many ways that an SAE could reconstruct points on that manifold.

Abstract

Recent work has proposed 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 of 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 strikingly interpretable examples, e.g. *circular* features representing days of the week and months of the year. We identify tasks where these exact circles are used to solve computational problems involving modular arithmetic in days of the week and months of the year. Next, we provide evidence that these circular features are indeed the fundamental unit of computation in these tasks with intervention experiments on Mistral 7B and Llama 3 8B, and we examine the continuity of the days of the week feature in Mistral 7B. Overall, our work argues that understanding multi-dimensional features is necessary to mechanistically decompose some model behaviors.

^{*}Joshua Engels, Eric J. Michaud, Isaac Liao, Wes Gurnee, and Max Tegmark. “Not all language model features are one-dimensionally linear.” *International Conference on Learning Representations* (2025).

9.1 Introduction

Language models trained for next-token prediction on large text corpora have demonstrated remarkable capabilities, including coding, reasoning, and in-context learning [1–4]. However, the specific algorithms models learn 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 by reverse-engineering trained models from the ground up into variables (features) and the programs (circuits) that process these variables [5].

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 [6] and circles [7,8], 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 [9–12]. These findings have led to the *linear representation hypothesis* (LRH): a hypothesis which has historically claimed both that 1. all representations in pretrained large language models lie along one-dimensional lines, and 2. model states are a simple sparse sum of these representations [12,13]. In this work, we specifically call into question the first part of the LRH.¹

For the most part, these two directions have been disconnected: [18] and [19] find intriguing hints of circular language model representations, and [12] speculate about the existence of feature manifolds, but these brief results only serve to further emphasize the lack of a unifying and satisfying perspective on the nature of language model features. In this work, we seek to bridge this gap by formalizing, investigating, and systematically searching for multi-dimensional language model features.

9.1.1 Contributions

1. In Section 9.3, we generalize the one-dimensional definition of a language model feature to multi-dimensional features and provide an updated multi-dimensional superposition hypothesis to account for these new features.
2. In Section 9.4, we build on the definitions proposed in section 9.3 to develop a theoretically grounded and empirically practical test that uses sparse autoencoders to find irreducible features. Using this test, we identify multi-dimensional representations automatically in GPT-2 and Mistral 7B, including circular representations for the day of the week and month of the year.
3. In Section 9.5, we show that Mistral 7B and Llama 3 8B use these circular representations when performing modular addition in days of the week and in months of the year. To the best of our knowledge, we are the first to find causal circular representations of concepts in a language model. We additionally find that the model’s circular representations respect a continuous notion of time.

¹An earlier version of this manuscript sparked discussion in the mechanistic interpretability community on the distinction between non-linear features and multi-dimensional features, and in fact this discussion directly led to a consensus around the two part LRH we describe above (see [14–17]). To clarify, we agree with these discussions, and believe the multi-dimensional features we find are "linear" in the sense that they are contained in a low-dimensional linear subspace, but "non-linear" in the sense that this low-dimensional subspace is not one-dimensional (and this is the sense we mean in the title).

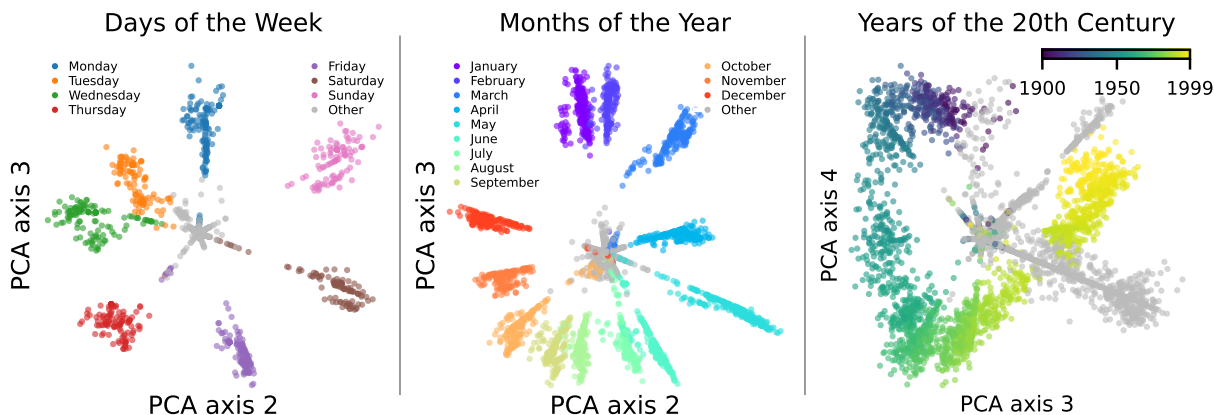


Figure 9.1: Circular representations of days of the week, months of the year, and years of the 20th century in layer 7 of GPT-2-small colored by the token they fire on. These representations were discovered via clustering SAE dictionary elements, described in Section 9.4. Points are colored according to the token which created the representation. See Figure 9.14 for other axes and Figure 9.15 for similar plots for Mistral 7B.

9.2 Related Work

Linear Representations: Early word embedding methods such as GloVe and Word2vec, although only trained using co-occurrence data, contained directions in their vector spaces corresponding to semantic concepts [20–22]. Recent research has found similar evidence of one-dimensional linear representations in sequence models trained only on next token prediction, including Othello board positions [23,24], the truth value of assertions [10], and numeric quantities such as longitude, latitude, birth year, and death year [9,11]. These results have inspired the linear representation hypothesis [13,25] defined above. [26] provide theoretical evidence for this hypothesis, assuming a latent (binary) variable-based model of language. Empirically, [12] and [27] successfully use sparse autoencoders to break down a model’s feature space into an over-complete basis of linear features. These works assume that the number of linear features stored in superposition exceeds the model dimensionality [25].

Multi-Dimensional Representations: There has been comparatively little research on multi-dimensional features in language models. [28] predict and verify that a transformer trained on a hidden Markov model uses a fractal structure to represent the probability of each next token, a clear example of a necessary multi-dimensional feature, but the analysis is restricted to a toy setting. [18,29] finds that GPT-2 learned position vectors form a helix, which implies a circle when “viewed” from below. Thus, we are not the first to find a circular feature in a language model. However, our work finds circular features that represent latent concepts from text, while the GPT-2 learned position vectors are specific to tokenization, separate from the rest of the model parameters, and causally implicated only due to positional attention masking. Another suggestive result, due to [30], is the presence of a U-shape in the representation of numbers between 0 and 100; however, [30] find that this representation is not causal, and they only show it exists within a specific prompt distribution. Recent work on dictionary learning [12] has speculated about multi-dimensional *feature manifolds*; our work is similar to this direction and develops the idea

of feature manifolds theoretically and empirically. Finally, in a separate direction, [31] argue 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.

Circuits: Circuits research seeks to identify and understand *circuits*, subsets of a model (usually represented as a directed acyclic graph) that explain specific behaviors [5]. The base units that form a circuit can be layers, neurons [5], or sparse autoencoder features [32]. In the first circuits-style work, [5] found line features that were combined into curve detection features in the InceptionV1 image model. More recent work has examined language models, for example the indirect object identification circuit in GPT-2 [33]. Given the difficulty of designing bespoke experiments, there has been increased research in automated circuit discovery methods [32,34,35].

Interpretability for Arithmetic Problems: [7] study models trained on modular arithmetic problems $a + b = c \pmod{m}$ and find that models that generalize well have circular representations for a and b . Further work by [8] and [36] shows that models use these circular representations to compute c via a “clock” algorithm and a separate “pizza” algorithm. 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 [30] and addition in GPT-J [37]. 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 by [19] investigates language models’ ability to increment numbers and finds linear features that fire on tokens equivalent modulo 10.

9.3 Definitions

This section focuses on hypotheses for how hidden states of language models can be decomposed into sums of functions of the input (features). We focus on L layer transformer models M that take in token input $\mathbf{t} = (t_1, \dots, t_n)$ from input token distribution $\mathcal{T}$, have hidden states $\mathbf{x}_{1,l}, \dots, \mathbf{x}_{n,l}$ for layers l , and output logit vectors $\mathbf{y}_1, \dots, \mathbf{y}_n$. Given a set of inputs T , we let $X_{i,l}$ be the set of all corresponding $\mathbf{x}_{i,l}$. We write matrices in capital bold, vectors and vector valued functions in lowercase bold, and sets in capital non-bold.

9.3.1 Multi-Dimensional Features

Definition 2 (Feature). We define a d_f -dimensional feature as a function $\mathbf{f}$ that maps a subset of the input space into $\mathbb{R}^{d_f}$. We say that a feature is *active* on the aforementioned subset.

The input token distribution $\mathcal{T}$ induces a d_f -dimensional probability distribution over feature vectors $\mathbf{f}(t)$. As an example, let $n = 1$ (so inputs are single tokens) and consider a feature $\mathbf{f}$ that maps integer tokens to their integer values in $\mathbb{R}^1$. Then $\mathbf{f}$ is a 1-dimensional feature that is active on integer tokens, and $\mathbf{f}(t)$ is the marginal integer occurrence distribution from the token distribution.

How can we differentiate “true” multi-dimensional features from sums of lower dimensional features? We make this distinction by examining the *reducibility* of a potential multi-dimensional feature. That is, $\mathbf{f}$ is a “true” multi-dimensional feature if it cannot be written as the sum of two

statistically independent features **and** it cannot be written as the sum of two non-co-occurring features. Formally, we have the following definition:

Definition 3. A feature $\mathbf{f}$ is *reducible* into features $\mathbf{a}$ and $\mathbf{b}$ if there exists an affine transformation

$$\mathbf{f} \mapsto \mathbf{R}\mathbf{f} + \mathbf{c} \equiv \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix} \quad (9.1)$$

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*, i.e., a sum of disjoint distributions, one of which is lower dimensional:
 $p(\mathbf{a}, \mathbf{b}) = wp(\mathbf{a})\delta(\mathbf{b}) + (1 - w)p(\mathbf{a}, \mathbf{b})$

Here, p is a probability density function that is conditional on the subset of $\mathcal{T}$ that $\mathbf{f}$ is active on, δ is the Dirac delta function, and $0 < w < 1$. By two probability distributions being disjoint, we mean that they have disjoint support (there is no set where both have positive probability measure, or equivalently the two features $\mathbf{a}$ and $\mathbf{b}$ cannot be active at the same time). In eq. (9.1), $\mathbf{a}$ is the first k components of the vector $\mathbf{R}\mathbf{f} + \mathbf{c}$ and $\mathbf{b}$ is the remaining $d_f - k$ 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 makes it separable or a mixture.

An 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 distribution². In natural language, a mixture might be a one hot encoding of “breed of dog”, while a separable distribution might be the “latitude” and “longitude” of location tokens.

In practice, the mixture and separability definitions may not be precisely satisfied. Thus, we soften our definitions to permit degrees of reducibility:

Definition 4 (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. (9.1):

$$S(\mathbf{f}) \equiv \min I(\mathbf{a}; \mathbf{b}) \quad (9.2)$$

where I denotes the mutual information. Smaller values of $S(\mathbf{f})$ mean that $\mathbf{f}$ is more separable.

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

$$M_\epsilon(\mathbf{f}) = \max_{\mathbf{v} \in \mathbb{R}^{d_f}, c \in \mathbb{R}} \mathbb{P}_{t \in \mathcal{T}} \left(|\mathbf{v} \cdot \mathbf{f}(t) + c| < \epsilon \sqrt{\mathbb{E}[(\mathbf{v} \cdot \mathbf{f}(t) + c)^2]} \right) \quad (9.3)$$

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

²since any multidimensional Gaussian can be rotated to have a diagonal covariance matrix

In section 9.7.2, we expand on the intuition behind why the separability and ϵ -mixture indices as defined here correspond to weakened versions of definition 3.

We develop optimization procedures to empirically solve for the separability and ϵ -mixture indices of two dimensional feature distributions. At a high level, the separability procedure iterates over a sweep of rotations and estimates the mutual information between the axes for each angle, while the ϵ -mixture index procedure performs gradient descent to find the ϵ band that contains the largest possible fraction of the feature distribution. For more details on the implementation of the tests, see section 9.7.2. In section 9.4, we apply these empirical tests to real language model feature distributions to find irreducible multi-dimensional features; we show the detailed test results on the “days of the week” cluster in fig. 9.2

9.3.2 Superposition

In this section, we propose an updated *superposition hypothesis* [25] that takes into account multi-dimensional features. First, we restate the original superposition hypothesis:

Definition 5 (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 $|\mathbf{x}_1 \cdot \mathbf{x}_2| \leq \delta$ 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 [25]). 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 $\mathbf{x}_{i,l}(t) = \sum_i \mathbf{v}_i f_i(t)$. We set $f_i(t)$ to zero when t is outside the domain of f_i .

In contrast, our new superposition hypothesis posits independence between irreducible multi-dimensional features instead of unknown levels of independence between one-dimensional features:

Hypothesis 2 (Multi-Dimensional 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 $\mathbf{V}_i \in \mathbb{R}^{d \times d_{f_i}}$ such that $\mathbf{x}_{i,l}(t) = \sum_i \mathbf{V}_i \mathbf{f}_i(t)$. We set $\mathbf{f}_i(t)$ to zero when t is outside the domain of $\mathbf{f}_i$.

Note that since multi-dimensional features can be written as the sums of projections of lower-dimensional features, our new superposition hypothesis is a stricter version of Hypothesis 1. In the next section, we will explore empirical evidence for our hypothesis, while in section 9.7.1, we prove upper and lower bounds on the number of δ -almost orthogonal matrices $\mathbf{V}_i$ that can be packed into d dimensional space.

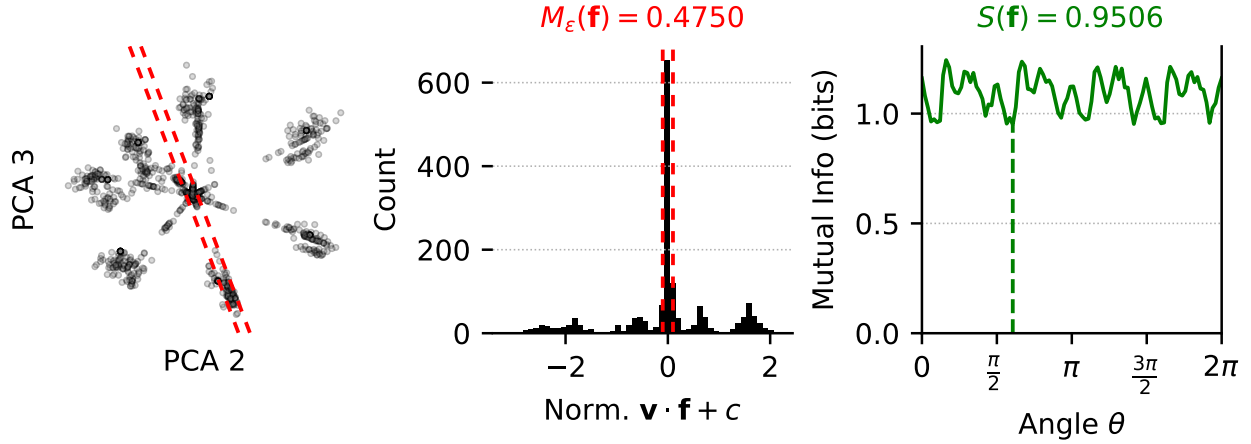


Figure 9.2: Empirical ϵ -mixture index and separability index for the “days of the week” cluster along PCA components 2 and 3. **Left:** The ϵ band parameterized by $\mathbf{v}$ and c that the optimization procedure found contained the highest fraction of points. **Mid:** Dot products of points in the feature distribution with the ϵ band; $M_\epsilon(\mathbf{f})$ is the percent of dot products within $\epsilon = 0.1$ of 0. **Right:** Estimated mutual information for different rotations of the space; $S(\mathbf{f})$ is the minimum over all rotations. This point cloud has a lower ϵ -mixture index and higher separability index than PCA projections within typical clusters (see fig. 9.3), indicating that it is more likely to be an irreducible multi-dimensional feature.

9.4 Sparse Autoencoders Find Multi-Dimensional Features

In this section, we describe a method to identify multi-dimensional features in language model hidden states using sparse autoencoders (SAEs). Sparse autoencoders (SAEs) deconstruct model hidden states into sparse vector sums from an over-complete basis [12,27]. For hidden states $X_{i,l}$, a one-layer SAE of size m with sparsity penalty λ minimizes the following dictionary learning loss [12,27]:

$$\text{DL}(X_{i,l}) = \underset{\mathbf{E} \in \mathbb{R}^{m \times d}, \mathbf{D} \in \mathbb{R}^{d \times m}}{\text{argmin}} \sum_{\mathbf{x}_{i,l} \in X_{i,l}} \left[\|\mathbf{x}_{i,l} - \mathbf{D} \cdot \text{ReLU}(\mathbf{E} \cdot \mathbf{x}_{i,l})\|_2^2 + \lambda \|\text{ReLU}(\mathbf{E} \cdot \mathbf{x}_{i,l})\|_0 \right] \quad (9.4)$$

In practice, the L_0 loss on the last term is relaxed to L_p for $0 < p \leq 1$ to make the loss differentiable. We call the m columns of $\mathbf{D}$ (vectors in $\mathbb{R}^d$) **dictionary elements**.

We now argue that SAEs can discover irreducible multi-dimensional features by *clustering* $\mathbf{D}$. We will consider a simple form of clustering: build a complete graph on $\mathbf{D}$ with edge weights equal to the cosine similarity between dictionary elements, prune all edges below a threshold T , and then set the clusters equal to the connected components of the graph. If we now consider the spaces spanned by each cluster, they will be approximately T -orthogonal by construction, since their basis vectors are all T -orthogonal. Now, consider some irreducible two-dimensional feature $\mathbf{f}$; we claim that if the SAE is large enough and $\mathbf{f}$ is active enough such that the SAE can reconstruct $\mathbf{f}$ when $\mathbf{f}$ is active, one of the clusters is likely to be exactly equal to $\mathbf{f}$. If $\mathbf{D}$ includes just two dictionary elements spanning $\mathbf{f}$, then these elements both must have nonzero activations post-ReLU to reconstruct $\mathbf{f}$ (otherwise $\mathbf{f}$ is a mixture). Because of the sparsity penalty in eq. (9.4),

this two-vector solution to reconstruct $\mathbf{f}$ is disincentivized, so instead the dictionary is likely to learn many elements that span $\mathbf{f}$. These dictionary elements will then have a high cosine similarity, and so the edges between them will not be pruned away during the clustering process; hence, they will be in a cluster.

Thus, we have a way to operationalize Hypothesis 2: clustering $\mathbf{D}$ finds T -orthogonal subspaces, and if irreducible multi-dimensional features exist, they are likely to be equal to some of these subspaces. 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. We use both the simple similarity-based pruning technique described above, as well as spectral clustering; see section 9.7.6 for details, including comments on scalability.

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. This step can be done manually by visually inspecting the PCA projections for known irreducible multi-dimensional structures (e.g. circles, see fig. 9.10) or automatically by passing the PCA projections to the tests for definition 4.

Pseudocode for this method is in the appendix in line 1. This method succeeds on toy datasets of synthetic irreducible multi-dimensional features; see section 9.7.4.³ We apply this method to language models using GPT-2 [38] SAEs trained by [39] for every layer and Mistral 7B [40] SAEs that we train on layers 8, 16, and 24 (training details in section 9.7.5).

Strikingly, we reconstruct irreducible multi-dimensional features that are *interpretable* circles: in GPT-2, days, months, and years are arranged circularly in order (see fig. 9.1); in Mistral 7B, days and months are arranged circularly in order (see fig. 9.15). These plots contain the PCA dimensions that most clearly show circular structure; these best dimensions are usually the second and third because the first PCA dim is an “intensity” direction that manifests as the radius of the circle in fig. 9.1 (thus the overall structure for these multi-d features is perhaps best thought of as a cone). See fig. 9.14 for all PCA dimensions visualized).

For each cluster of GPT-2 SAE features, we take the reconstructed activations and project them onto PCA components 1-2, 2-3, 3-4, and 4-5 (or fewer if there are fewer features in the cluster) and measure the separability index and ϵ -mixture index of each 2D point cloud as described in section 9.7.2. The mean scores across these planes are a computationally tractable approximation of definition 4. We plot these mean scores in fig. 9.3, and find that the features which we had manually identified in fig. 9.1 are among the top scoring features along both measures of irreducibility.

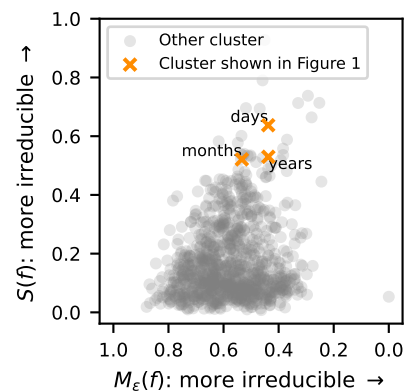


Figure 9.3: Mixture index and separability index of GPT-2 features. Features from fig. 9.1, which we had manually identified, score highly as candidate multidimensional features with these metrics.

³Code: <https://github.com/JoshEngels/MultiDimensionalFeatures>

Thus, our theoretical tests can indeed be used to find interpretable irreducible features. We show the top 20 feature clusters, measured by the product of $(1 - \epsilon\text{-mixture index})$ and separability index, in section 9.7.7. Out of all 1000 clusters, the fig. 9.1 clusters rank 9, 28, and 15 by this metric, respectively.⁴

9.5 Circular Representations in Large Language Models

In this section, we examine tasks in which models *use* the multi-dimensional features we discovered in section 9.4, thereby providing evidence that these representations are indeed the fundamental unit of computation for some problems. Inspired by prior work studying circular representations in modular arithmetic [7], we define two prompts that represent “natural” modular arithmetic tasks:

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 [41] achieve reasonable performance on the Weekdays task and excellent performance on the Months task (measured by comparing the highest logit valid token against the ground truth answer), as summarized in table 9.1. Interestingly, although these problems are equivalent to modular arithmetic problems $\alpha + \beta \equiv ? \pmod{m}$ for $m = 7, 12$, both models get trivial accuracy on plain modular addition prompts, e.g. “ $5+3 \pmod{7} \equiv$ ”. Finally, although GPT-2 has circular representations, it gets trivial accuracy on Weekdays and Months.

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 j th prompt asks about α_j , β_j , and γ_j .

⁴We also tried an alternative ranking scheme: we sorted the clusters by separability and irreducibility and set the cluster score equal to the minimum sorted position between the two sorted lists. The fig. 9.1 clusters rank 8, 105, and 12 by this metric.

Table 9.1: Aggregate model accuracy on days of the week and months of the year modular arithmetic tasks. Performance broken down by problem instance in section 9.7.9.

Model	Weekdays	Months
Llama 3 8B	29 / 49	143 / 144
Mistral 7B	31 / 49	125 / 144
GPT-2	8 / 49	10 / 144

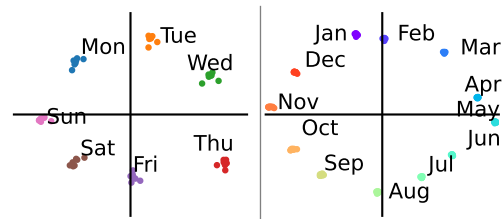


Figure 9.4: Top two PCA components on the α token. Colors show α . **Left:** Layer 30 of Mistral on Weekdays. **Right:** Layer 5 of Llama on Months.

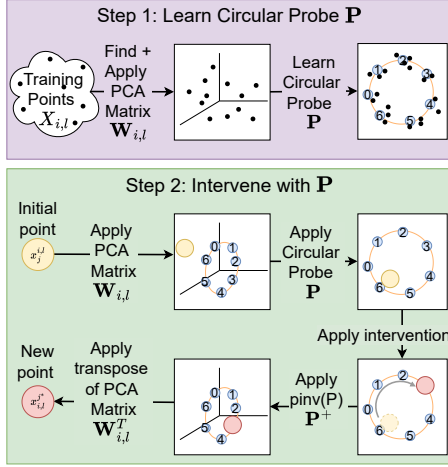


Figure 9.5: Visual representation of the circular intervention process. **Top:** We learn a circular probe on the PCA projection of a training set. **Bot:** To intervene, we change the circular representation to α'_j and average ablate other dimensions.

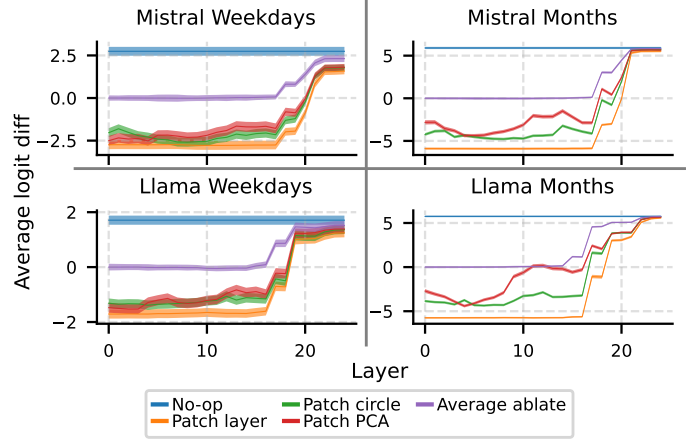


Figure 9.6: Mean and 96% error bars for intervening on the α token across layers using different intervention methods. The circular intervention technique outperforms patching only the top 5 PCA components and leaving the rest unchanged, and almost reaches the upper bound performance of patching the entire layer.

We confirm that Llama 3 8B and Mistral 7B have circular representations of α on this task by examining the PCA projections of hidden states across prompts at various layers on the α token. We plot two of these in fig. 9.4 and show all layers in fig. 9.18. These plots show circular representations as the highest varying two components in the model’s representation of α at many layers.

9.5.1 Intervening on Circular Day and Month Representations

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 [42]. 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. 9.5):

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 $\mathbf{W}_{i,l} \cdot \mathbf{x}_{i,l}^j$ to a unit circle in α . In other words, if $\text{circle}(\alpha) = [\cos(2\pi\alpha/7), \sin(2\pi\alpha/7)]$ for Weekdays and

$\text{circle}(\alpha) = [\cos(2\pi\alpha/12), \sin(2\pi\alpha/12)]$ for Months, $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \underset{\mathbf{P}' \in \mathbb{R}^{2,k}}{\operatorname{argmin}} \sum_{\mathbf{x}_{i,l}^j} \left\| \mathbf{P}' \cdot \mathbf{W}_{i,l} \cdot \mathbf{x}_{i,l}^j - \text{circle}(\alpha) \right\|_2^2 \quad (9.5)$$

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 $\mathbf{P} \cdot \mathbf{W}_{i,l}$, which will be close to $\text{circle}(\alpha_j)$, with the “clean” point $\text{circle}(\alpha_{j'})$. Note that we do not use the hidden state $\mathbf{x}_{i,l}^{j'}$ from the “clean” run, only the “clean” label $\alpha_{j'}$. In practice, other subspaces of $\mathbf{x}_{i,l}^j$ may be used concurrently by the model in “backup” circuits (see e.g. [33]) to compute the answer, so if we just intervene on the circular subspace the remaining components of the activation may interfere in downstream computations. Thus, to isolate the effect of our intervention, we set the average ablate the portion of the activation 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 and $\mathbf{P}^+$ be the pseudoinverse of $\mathbf{P}$, we intervene via the formula

$$\mathbf{x}_{i,l}^{j*} = \overline{\mathbf{x}_{i,l}} + \mathbf{W}_{i,l}^T \mathbf{P}^+ (\text{circle}(\alpha_{j'}) - \overline{\mathbf{x}_{i,l}}) \quad (9.6)$$

The main takeaway from fig. 9.6 is that circular subspaces are causally implicated in computing γ , especially for Weekdays. Across all models and tasks, early layer interventions on the circular subspace have almost the same intervention effect as patching the entire layer, and are usually better than patching the top PCA dimensions from the clean problem.

Patching experiments in section 9.7.10 show α is copied to the final token on layers 15 to 17, which is why interventions drop off there. Additionally, while in this section we train a probe on a dataset of prompts, in section 9.5.2, we show that intervening on the circle discovered via SAE clustering in section 9.4 also works.

We run our patching on 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 experiments for Weekdays and $144 * 11$ patching experiments for Months. We also run baselines 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, and (3) replace the entire layer with the average across the task. The metric we use is *average logit difference* across all patching experiments between the original correct token (α_j) and the target token ($\alpha_{j'}$). See fig. 9.6 for these interventions on all layers of Mistral 7B and Llama 3 8B on Weekdays and Months.

To investigate exactly how models use the circular subspace, we perform *off distribution* interventions. We modify eq. (9.6) so that instead of intervening on the circumference $\text{circle}(\alpha)$,

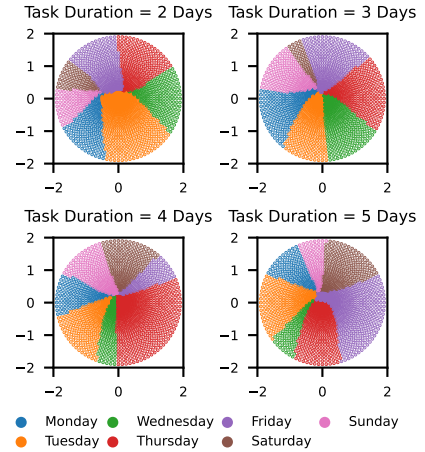


Figure 9.7: Off distribution interventions on Mistral layer 5 on the Weekdays task. The color corresponds to the highest logit γ after performing the circular subspace intervention on that point.

we sweep over a grid of positions (r, θ) within the circle:

$$\mathbf{x}_{i,l}^{j*} = \overline{\mathbf{x}_{i,l}} + \mathbf{W}_{i,l}^T \mathbf{P}^+ [r \cos(\theta), r \sin(\theta)]^T - \overline{\mathbf{x}_{i,l}} \quad (9.7)$$

We intervene with $r \in [0, 0.1, \dots, 2]$, $\theta \in [0, 2\pi/100, \dots, 198\pi/100]$ and record the highest logit γ after the forward pass. fig. 9.7 displays these results on Mistral layer 5 for $\beta \in [2, 3, 45]$. They imply that Mistral treats the circle as a multi-dimensional representation with α encoded in the angle.

9.5.2 Intervening with the SAE Plane

In the last section, we train probes manually on the PCA of the activations to fit a circle. A perhaps more natural approach is intervening with the precise circle we found in section 9.4. To determine if this approach is feasible, we first project layer 8 Mistral 7B Weekdays activations into the weekdays plane that was discovered by clustering (see fig. 9.15; the plane is defined by PCA dimensions 2 and 3 of the cluster). In the rest of this section, we call this plane the SAE plane. In fig. 9.21, we find that indeed, the Weekdays representations projected into the SAE plane form a circle (see fig. 9.21). We thus can fit a circular probe to this 2D plane as in eq. (9.5). Similarly, we call this probe the SAE probe.

Because we only have layer 8 clustering results for Mistral, we train an SAE probe only on layer 8. We then evaluate interventions using this SAE probe on layer 8 of Mistral, but also at neighboring layers, since nearby layers should have similar representations (see e.g. [43]). We compare to two baselines: 1) training a normal circular probe on the PCA projections of each layer as described in section 9.5.1, and 2) training a circular probe only on layer 8 and then evaluating on adjacent layers (in the same way as for the layer 8 SAE probe). We show the results of these methods in fig. 9.8.

We find that on all layers, using the SAE probe only slightly decreases intervention performance as compared to training a circular probe (from -2.58 to -2.01 average logit difference on layer 8). Even more interestingly, the layer 8 SAE probe is much more robust to layer shifts than the layer 8 circular probe; for example, using the layer 8 circular probe on layer 6 results in an average logit difference of 0.029, whereas using the layer 8 SAE probe results in an average logit difference of -2.32. This is intriguing evidence that the SAE is perhaps finding more “true” (or at least more robust) features than our circular probing technique.

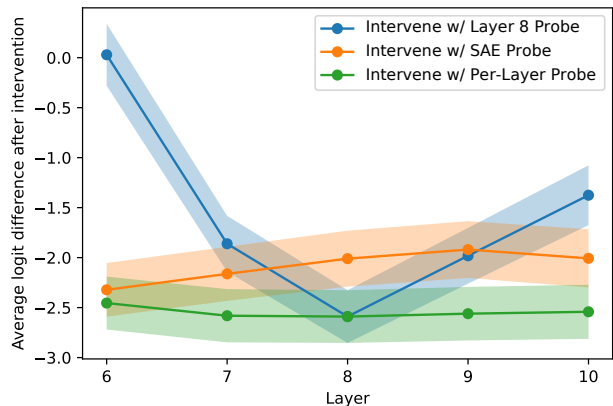


Figure 9.8: Interventions on the Mistral 7B Weekdays task with different methods of determining the probe.

9.5.3 Continuity of Circular Representations

In past sections, the representations of the interpretable numeric quantities we have discovered have been mostly *discontinuous*; that is, the days of the week and months of the year in fig. 9.1 and fig. 9.15 are clustered at the vertices of a heptagon and dodecagon, and there is nothing "between" adjacent weekdays or months along the circle. In this section, we will examine the "continuity" of the circular features we have discovered. Although continuity of the representation is not a requirement of definition 4, it would further decrease the ϵ -mixture index, and would also increase our subjective perception of the circular feature as an intrinsic model feature representing a continuous quantity (time). Thus, we create a synthetic dataset containing the text "[very early/very late] on [Monday/Tuesday/.../Sunday]" and simply plot the projections of the layer 30 activations into the top two PCA components of the activations of [Monday/Tuesday/.../Sunday]. The results, shown in fig. 9.9, show that Mistral 7B indeed can map intermediate quantities to their expected place in the circle: the very early and very late version of each weekday are more towards the last and the next weekday along the circle, respectively. We show similar results for "[morning/evening] on [Monday/Tuesday/.../Sunday]" in Appendix fig. 9.22.

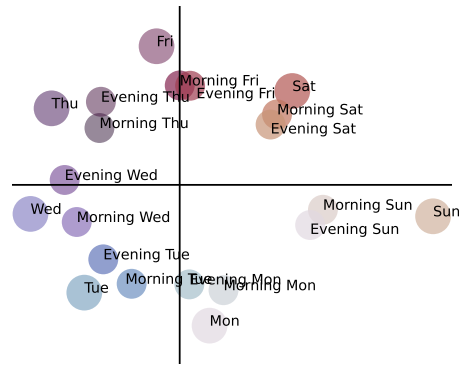


Figure 9.9: Layer 30 Mistral 7B activations for [morning/evening] on [Monday/Tuesday/.../Sunday], plotted projected into the PCA plane for [Monday/Tuesday/.../Sunday].

9.6 Discussion

Our work proposes a significant refinement to the simple one-dimensional linear representation hypothesis. While previous work has convincingly shown the existence of one-dimensional features, we find evidence for irreducible multi-dimensional representations, requiring us to generalize the notion of a feature to higher dimensions. Fortunately, we find that existing unsupervised feature extraction methodologies like sparse autoencoders can readily be applied to discover multi-dimensional representations. However, we think our work raises interesting questions about whether individual SAE features are appropriate “mediators” [44] for understanding model computation, if some features are in fact multi-dimensional. Although taking a multi-dimensional representation perspective may be more complicated, we believe that uncovering the true (perhaps multi-dimensional) nature of model representations is necessary for discovering the underlying algorithms that use these representations. Ultimately, our field aims to turn complex circuits in future more-capable models into formally verifiable programs [45,46], which requires the ground truth “variables” of language models; we believe this work takes an important step towards discovering these variables.

Limitations: It is unclear why we did not find more interpretable multi-dimensional features. We are unsure if we are failing to interpret some of the high-scoring multi-dimensional features, if most multi-dimensional features lie in dimensions higher than two, if our clustering technique is not powerful enough to find some features, or if there are truly not that many. Additionally, our definitions for irreducible features (definition 3) are purely statistical and not intervention

based, and also had to be relaxed to hold in practice, resulting in measures that return a possibly subjective “degree” of reducibility (definition 4). Thus, although this work provides preliminary evidence for the multi-dimensional superposition hypothesis (Hypothesis 2), it is still unclear if this theory provides the best description for the representations models use. Future work might make progress on this question by investigating new techniques for decomposing model representations, exploring higher dimensional representations, or determining conclusively whether models use representations in ways that necessitate the representations are non-linear.

Acknowledgements

We thank (in alphabetical order) Dowon Baek, Kaivu Hariharan, Vedang Lad, Ziming Liu, and Tony Wang for helpful discussions 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

- [1] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, et al. “Sparks of artificial general intelligence: Early experiments with gpt-4.” *arXiv preprint arXiv:2303.12712*, 2023.
- [2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Al-tenschmidt, S. Altman, S. Anadkat, et al. “Gpt-4 technical report.” *arXiv preprint arXiv:2303.08774*, 2023.
- [3] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Tech. rep. Anthropic, 2024.
- [4] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, et al. “Gemini: a family of highly capable multimodal models.” *arXiv preprint arXiv:2312.11805*, 2023.
- [5] C. Olah, N. Cammarata, L. Schubert, G. Goh, M. Petrov, and S. Carter. “Zoom In: An Introduction to Circuits.” *Distill*, 2020. <https://distill.pub/2020/circuits/zoom-in>. DOI: [10.23915/distill.00024.001](https://doi.org/10.23915/distill.00024.001).
- [6] E. J. Michaud, I. Liao, V. Lad, Z. Liu, A. Mudide, C. Loughridge, Z. C. Guo, T. R. Kheirkhah, M. Vukelić, and M. Tegmark. “Opening the AI black box: program synthesis via mechanistic interpretability.” *arXiv preprint arXiv:2402.05110*, 2024.
- [7] Z. Liu, O. Kitouni, N. S. Nolte, E. Michaud, M. Tegmark, and M. Williams. “Towards understanding grokking: An effective theory of representation learning.” *Advances in Neural Information Processing Systems*, 35, 2022, pp. 34651–34663.
- [8] N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. “Progress measures for grokking via mechanistic interpretability.” *arXiv preprint arXiv:2301.05217*, 2023.
- [9] W. Gurnee and M. Tegmark. “Language models represent space and time.” *arXiv preprint arXiv:2310.02207*, 2023.

- [10] S. Marks and M. Tegmark. “The geometry of truth: Emergent linear structure in large language model representations of true/false datasets.” *arXiv preprint arXiv:2310.06824*, 2023.
- [11] B. Heinzerling and K. Inui. “Monotonic Representation of Numeric Properties in Language Models.” *arXiv preprint arXiv:2403.10381*, 2024.
- [12] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. URL: <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [13] K. Park, Y. J. Choe, and V. Veitch. “The linear representation hypothesis and the geometry of large language models.” *arXiv preprint arXiv:2311.03658*, 2023.
- [14] C. Olah. “What is a Linear Representation? What is a Multidimensional Feature?” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/july-update/index.html#linear-representations>.
- [15] R. Csordás, C. Potts, C. D. Manning, and A. Geiger. “Recurrent Neural Networks Learn to Store and Generate Sequences using Non-Linear Representations.” *arXiv preprint arXiv:2408.10920*, 2024.
- [16] J. Mendel. “SAE feature geometry is outside the superposition hypothesis.” *AI Alignment Forum*, 2024. URL: <https://www.alignmentforum.org/posts/MFBTjb2qf3ziWmzz6/sae-feature-geometry-is-outside-the-superposition-hypothesis>.
- [17] S. Kantamneni and M. Tegmark. “Language Models Use Trigonometry to Do Addition.” *arXiv preprint arXiv:2502.00873*, 2025.
- [18] A. Yedidia. *GPT-2’s Positional Embedding Matrix is a Helix*. Accessed: 2024-09-16. 2023. URL: <https://www.lesswrong.com/posts/qvWP3aBDBaqXvPNhS/gpt-2-s-positional-embedding-matrix-is-a-helix>.
- [19] R. Gould, E. Ong, G. Ogden, and A. Conmy. “Successor heads: Recurring, interpretable attention heads in the wild.” *arXiv preprint arXiv:2312.09230*, 2023.
- [20] T. Mikolov, W.-t. Yih, and G. Zweig. “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*. 2013, pp. 746–751.
- [21] J. Pennington, R. Socher, and C. D. Manning. “Glove: Global vectors for word representation.” In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.
- [22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. “Distributed representations of words and phrases and their compositionality.” *Advances in neural information processing systems*, **26**, 2013.
- [23] N. Nanda, A. Lee, and M. Wattenberg. “Emergent linear representations in world models of self-supervised sequence models.” *arXiv preprint arXiv:2309.00941*, 2023.
- [24] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. “Emergent world representations: Exploring a sequence model trained on a synthetic task.” *arXiv preprint arXiv:2210.13382*, 2022.

- [25] N. Elhage et al. “Toy Models of Superposition.” *Transformer Circuits Thread*, 2022. https://transformer-circuits.pub/2022/toy_model/index.html.
- [26] Y. Jiang, G. Rajendran, P. Ravikumar, B. Aragam, and V. Veitch. “On the Origins of Linear Representations in Large Language Models.” *arXiv preprint arXiv:2403.03867*, 2024.
- [27] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey. “Sparse autoencoders find highly interpretable features in language models.” *arXiv preprint arXiv:2309.08600*, 2023.
- [28] A. Shai, P. Riechers, L. Teixeira, A. Oldenziel, and S. Marzen. *Transformers Represent Belief State Geometry in their Residual Stream*. <https://www.alignmentforum.org/posts/gTZ2SxesbHckJ3CkF/transformers-represent-belief-state-geometry-in-their>. 2024.
- [29] A. Yedidia. *The positional embedding matrix and previous-token heads: how do they actually work?* Accessed: 2024-09-17. 2023. URL: <https://www.lesswrong.com/posts/zRA8B2FJLtTYRgie6/the-positional-embedding-matrix-and-previous-token-heads-how>.
- [30] M. Hanna, O. Liu, and A. Variengien. “How does gpt-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model.” *Advances in Neural Information Processing Systems*, 36, 2024.
- [31] S. Black, L. Sharkey, L. Grinsztajn, E. Winsor, D. Braun, J. Merizian, K. Parker, C. R. Guevara, B. Millidge, G. Alfour, et al. “Interpreting neural networks through the polytope lens.” *arXiv preprint arXiv:2211.12312*, 2022.
- [32] S. Marks, C. Rager, E. J. Michaud, Y. Belinkov, D. Bau, and A. Mueller. “Sparse Feature Circuits: Discovering and Editing Interpretable Causal Graphs in Language Models.” *arXiv preprint arXiv:2403.19647*, 2024.
- [33] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt. “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small.” *arXiv preprint arXiv:2211.00593*, 2022.
- [34] A. Conmy, A. Mavor-Parker, A. Lynch, S. Heimersheim, and A. Garriga-Alonso. “Towards automated circuit discovery for mechanistic interpretability.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 16318–16352.
- [35] A. Syed, C. Rager, and A. Conmy. “Attribution Patching Outperforms Automated Circuit Discovery.” *arXiv preprint arXiv:2310.10348*, 2023.
- [36] Z. Zhong, Z. Liu, M. Tegmark, and J. Andreas. “The clock and the pizza: Two stories in mechanistic explanation of neural networks.” *Advances in Neural Information Processing Systems*, 36, 2024.
- [37] A. Stolfo, Y. Belinkov, and M. Sachan. “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*. 2023, pp. 7035–7052.
- [38] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. “Language models are unsupervised multitask learners.” *OpenAI blog*, 2019.
- [39] J. Bloom. *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>. 2024.

- [40] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. I. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. “Mistral 7B.” *arXiv preprint arXiv:2310.06825*, 2023.
- [41] AI@Meta. *Llama 3 Model Card*. 2024. URL: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.
- [42] F. Zhang and N. Nanda. “Towards best practices of activation patching in language models: Metrics and methods.” *arXiv preprint arXiv:2309.16042*, 2023.
- [43] N. Belrose, Z. Furman, L. Smith, D. Halawi, I. Ostrovsky, L. McKinney, S. Biderman, and J. Steinhardt. “Eliciting latent predictions from transformers with the tuned lens.” *arXiv preprint arXiv:2303.08112*, 2023.
- [44] A. Mueller, J. Brinkmann, M. Li, S. Marks, K. Pal, N. Prakash, C. Rager, A. Sankaranarayanan, A. S. Sharma, J. Sun, et al. “The quest for the right mediator: A history, survey, and theoretical grounding of causal interpretability.” *arXiv preprint arXiv:2408.01416*, 2024.
- [45] M. Tegmark and S. Omohundro. “Provably safe systems: the only path to controllable agi.” *arXiv preprint arXiv:2309.01933*, 2023.
- [46] D. Dalrymple, J. Skalse, Y. Bengio, S. Russell, M. Tegmark, S. Seshia, S. Omohundro, C. Szegedy, B. Goldhaber, N. Ammann, et al. “Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems.” *arXiv preprint arXiv:2405.06624*, 2024.
- [47] W. B. Johnson and J. Lindenstrauss. “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, 1984, pp. 189–206. ISBN: 0-8218-5030-X. DOI: [10.1090/conm/026/737400](https://doi.org/10.1090/conm/026/737400).
- [48] N. J. Higham. *Singular Value Inequalities*. <https://nhigham.com/2021/05/04/singular-value-inequalities/>. May 2021.
- [49] S. A. Gershgorin. “Über die Abgrenzung der Eigenwerte einer Matrix.” *Izvestiya Rossiiskoi akademii nauk. Seriya matematicheskaya*, (6), 1931, pp. 749–754.
- [50] B. J. (<https://mathoverflow.net/users/2554/bill-johnson>). *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>.
- [51] N. Alon. “Problems and results in extremal combinatorics—I.” *Discrete Mathematics*, 273(1-3), 2003, pp. 31–53.
- [52] D. Morwani, B. L. Edelman, C.-A. Oncescu, R. Zhao, and S. Kakade. “Feature emergence via margin maximization: case studies in algebraic tasks.” *arXiv preprint arXiv:2311.07568*, 2023.
- [53] S. Foucart and H. Rauhut. *A Mathematical Introduction to Compressive Sensing*. 2013.
- [54] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, et al. “The pile: An 800gb dataset of diverse text for language modeling.” *arXiv preprint arXiv:2101.00027*, 2020.
- [55] B. Peng, C. Li, P. He, M. Galley, and J. Gao. “Instruction tuning with gpt-4.” *arXiv preprint arXiv:2304.03277*, 2023.

- [56] S. Lattanzi, T. Lavastida, K. Lu, and B. Moseley. “A framework for parallelizing hierarchical clustering methods.” In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2019, Würzburg, Germany, September 16–20, 2019, Proceedings, Part I*. Springer. 2020, pp. 73–89.
- [57] N. Nanda and J. Bloom. *TransformerLens*. <https://github.com/TransformerLensOrg/TransformerLens>. 2022.

9.7 Appendix

9.7.1 Multi-Dimensional Feature Capacity

The Johnson-Lindenstrauss (JL) Lemma [47] implies that we can choose $e^{Cd\delta^2}$ pairwise one-dimensional δ -orthogonal vectors to satisfy Hypothesis 1 for some constant C , thus allowing us to build the model’s feature space with a number of one-dimensional δ -orthogonal features exponential in d . We now prove a similar result for low-dimensional projections (the main idea of the proof is to combine δ -orthogonal vectors as guaranteed from the JL lemma):

Theorem 1. *For any d' and δ , it is possible to choose $\frac{1}{d_{\max}} e^{C_1(d/d'^2)\delta^2}$ pairwise δ -orthogonal matrices $\mathbf{A}_i \in \mathbb{R}^{n_i \times d'}$ for some constant C_1 . Furthermore, it is not possible to choose more than $e^{C_2(d-d_{\max}\delta \log(\frac{1}{\delta}))}$ for some constant C_2 .*

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

Lemma 1. *Pick n pairwise δ -orthogonal unit vectors in $\mathbf{v}_1, \dots, \mathbf{v}_n \in \mathbb{R}^d$. Let $\mathbf{y} \in \mathbb{R}^d$ be a unit norm vector that is a linear combination of unit norm vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ with coefficients $z_1, \dots, z_n \in \mathbb{R}$. We can write $\mathbf{A} = [\mathbf{v}_1, \dots, \mathbf{v}_n]$ and $\mathbf{z} = [z_1, \dots, z_n]^T$, so that we have $\mathbf{y} = \sum_{k=1}^n z_k \mathbf{v}_k = \mathbf{A}\mathbf{z}^T$ with $\|\mathbf{y}\|_2 = 1$. Then,*

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

Proof. We will first bound the L_2 norm of $\mathbf{z}$. If σ_n is the minimum singular value of $\mathbf{A}$, then we have via standard singular value inequalities [48]

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

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 [49], 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 $\|\mathbf{z}\|_2$, we have that $\|\mathbf{z}\|_2 \leq 1/\sqrt{1 - \delta(n-1)}$. 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 $\sqrt{nr}$, so

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

□

Theorem 1. For any d' and δ , it is possible to choose $\frac{1}{d_{\max}} e^{C_1(d/d'^2)\delta^2}$ pairwise δ -orthogonal matrices $\mathbf{A}_i \in \mathbb{R}^{n_i \times d'}$ for some constant C_1 . Furthermore, it is not possible to choose more than $e^{C_2(d-d_{\max}\delta \log(\frac{1}{\delta}))}$ for some constant C_2 .

Proof. By the JL lemma [47,50], for any d and δ , we can choose $e^{Cd\delta^2}$ δ -orthogonal unit vectors in $\mathbb{R}^d$ indexed as $\mathbf{v}_i$, for some constant C . Let $\mathbf{A}_i = [\mathbf{v}_{d_{\max}*i}, \dots, \mathbf{v}_{d_{\max}*i+n_i-1}]$ 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 $\frac{1}{d_{\max}} e^{Cd\delta^2}$ matrices $\mathbf{A}_i$.

Now, consider two of these matrices $\mathbf{A}_i = [\mathbf{v}_1, \dots, \mathbf{v}_{n_i}]$ and $\mathbf{A}_j = [\mathbf{u}_1, \dots, \mathbf{u}_{n_j}]$, $i \neq j$; 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 $\mathbf{y}_j$:

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

Thus $\mathbf{A}_i$ and $\mathbf{A}_j$ are $f(\delta)$ -orthogonal for $f(\delta) = \delta d_{\max} / (1 - \delta d_{\max})$, and so it is possible to choose $\frac{1}{d_{\max}} e^{Cd\delta^2}$ 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 $\frac{1}{d_{\max}} e^{Cd\delta^2 / ((1+\delta)^2 d_{\max}^2)}$ pairwise

δ -orthogonal projection matrices. Because $1 + \delta$ is at most 2 with $\delta \in (0, 1)$, we can further simplify the exponent and find that it is possible to choose $\frac{1}{d_{\max}} e^{C(d/d_{\max}^2)\delta^2/4}$ pairwise δ -orthogonal projection matrices. Absorbing the 4 into the constant C finishes the proof of the lower bound.

For the upper bound, we can proceed much more simply. Consider k pairwise δ -orthogonal matrices $A_i \in \mathbb{R}^{d'}$. Since these matrices are full rank, their column spaces each parameterize a subspace of dimension d' , and so by a result from [51] it is possible to choose $e^{Cd'\delta^2 \log(\frac{1}{\delta})}$ almost orthogonal vectors in this subspace. Furthermore, by our definition of δ -orthogonal matrices, all pairs of these vectors between subspaces will be δ -orthogonal. Finally, again by [51] we cannot have more than $e^{Cd\delta^2 \log(\frac{1}{\delta})}$ δ -orthogonal vectors overall, so we have that

$$ke^{Cd_{\max}\delta^2 \log(\frac{1}{\delta})} < e^{Cd\delta^2 \log(\frac{1}{\delta})}$$

and simplifying,

$$k < e^{C(d-d_{\max})\delta^2 \log(\frac{1}{\delta})}$$

□

These results imply that models can still represent an exponential number of higher dimensional features. However, there is a large exponential gap between the lower and upper bound we have shown. If the lower bound is reasonably tight, then this would mean that models would be highly incentivized to fit features within the smallest dimensional space possible, suggesting a reason for recent work showing interesting compressed encodings of multi-dimensional features in toy problems [52].

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.

Finally, we note that the dictionary learning literature may have discovered similar results in the past (which we were unaware of), see [53].

9.7.2 More on Reducibility

Additional Intuition for Definitions

Here, we present some extra intuition and high level ideas for understanding our definitions and the motivation behind them. Roughly, we intend for our definitions in the main text to identify representations in the model that describe an object or concept in a way that fundamentally takes multiple dimensions. We operationalize this as finding a subspace of representations that 1. has basis vectors that “always co-occur” no matter the orientation 2. is not made up of combinations of independent lower-dimensional features.

1. The first condition is met by the mixture part of our definition. The feature in question should be part of an irreducible manifold, and so should “fill” a plane or hyperplane. There shouldn’t be any part of the plane where the probability distribution of the feature is concentrated, because this region is then likely part of a lower dimensional feature. The idea of this part of the definition is to capture multi-dimensional objects; if the entire multi-dimensional space is truly

being used to represent a high-dimensional object, then the representations for the object should be “spread out” entirely through the space.

2. The second condition is met by the separability part of our definition. This part of the definition is intended to rule out features that co-occur frequently but are fundamentally not describing the same object or concept. For example, latitude and longitude are not a mixture in that they frequently co-occur, but we do not think it is necessarily correct to say they are part of the same multi-dimensional feature because they are independent.

Empirical Irreducible Feature Test Details

Our tests for reducibility require the computation of two quantities $S(\mathbf{f})$ for the separability index and $M_\epsilon(\mathbf{f})$ for the ϵ -mixture index. We describe how we compute each index in the following two subsections.

Separability Index

We define the separability index in Equation 9.2 as

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

where the min is over rotations $\mathbf{R}$ used to split $\mathbf{f}' = \mathbf{R}\mathbf{f} + \mathbf{c}$ into $\mathbf{a}$ and $\mathbf{b}$. In two dimensions, the rotation is defined by a single angle, so we can iterate over a grid of 1000 angles and estimate the mutual information between $\mathbf{a}$ and $\mathbf{b}$ for each angle. We first normalize $\mathbf{f}$ by subtracting off the mean and then dividing by the root mean squared norm of $\mathbf{f}$ (and multiplying by $\sqrt{2}$ since the toy datasets are in two dimensions). To estimate the mutual information, we first clip the data $\mathbf{f}$ to a 6 by 6 square centered on the origin. We then bin the points into a 40 by 40 grid, to produce a discrete distribution $p(a, b)$. After computing the marginals $p(a)$ and $p(b)$ by summing the distribution over each axis, we obtain the mutual information via the formula

$$I(\mathbf{a}; \mathbf{b}) = \sum_{a,b} p(a, b) \log \frac{p(a, b)}{p(a)p(b)} \quad (9.8)$$

ϵ -Mixture Index

We define the ϵ -mixture index in Equation 9.3 as

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

The challenge with computing $M_\epsilon(\mathbf{f})$ is to compute the maximum. We opted to maximize via gradient descent; and we guaranteed differentiability by softening the inequality $<$ with a sigmoid,

$$M_{\epsilon,T}(\mathbf{f}, \mathbf{v}, c) = \mathbb{E} \left(\sigma \left(\frac{1}{T} \left(\epsilon - \frac{|\mathbf{v} \cdot \mathbf{f} + c|}{\sqrt{\mathbb{E}[(\mathbf{v} \cdot \mathbf{f} + c)^2]}} \right) \right) \right) \quad (9.9)$$

where T is a temperature, which we linearly decay from 1 to 0 throughout training. We optimize for $\mathbf{v}$ and c using this loss $M_{\epsilon,T}(\mathbf{f}, \mathbf{v}, c)$ using full batch gradient descent over 10000 steps with

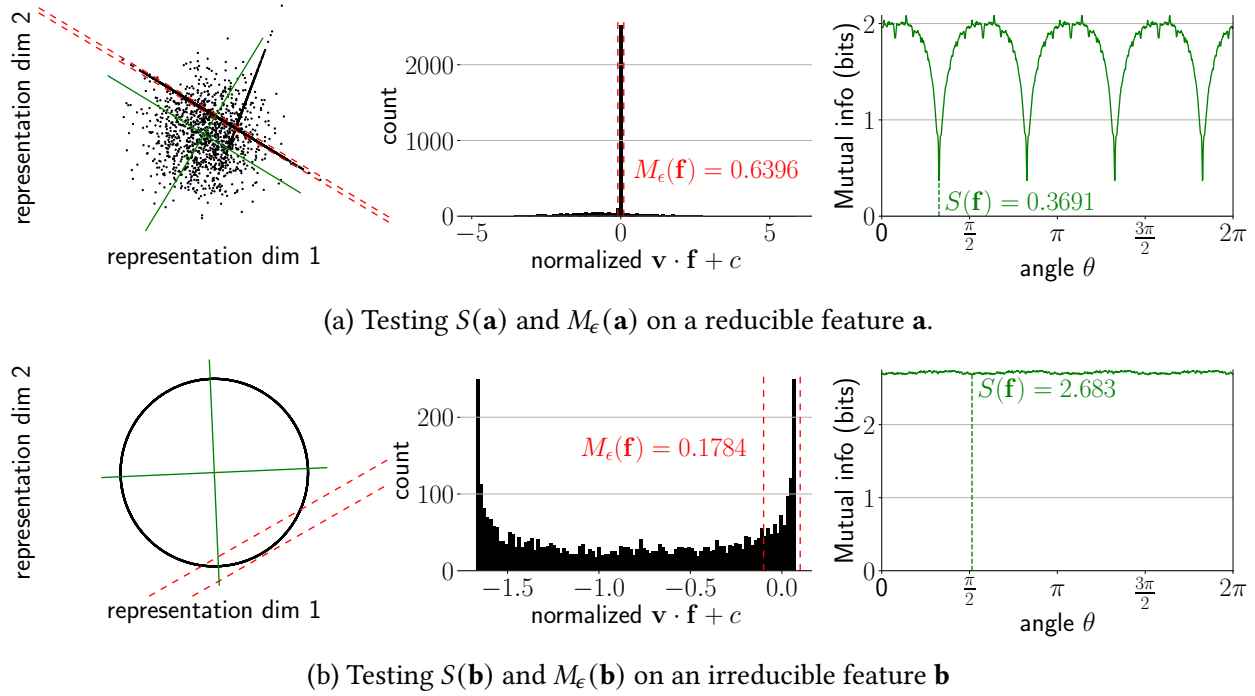


Figure 9.10: Testing irreducibility of synthetic features. **Left in each subfigure:** Distributions of $\mathbf{x}$. For feature $\mathbf{a}$, 63.96% lies within the narrow dotted lines, indicating the feature is likely a mixture. For feature $\mathbf{b}$, 17.84% lies within the wide lines, indicating the feature is unlikely to be a mixture. The green cross indicates the angle θ that minimizes mutual information. **Middle 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:** Mutual information between $\mathbf{a}$ and $\mathbf{b}$ as a function of the rotation angle θ of matrix $\mathbf{R}$. Feature $\mathbf{b}$ has a large minimum mutual information so is unlikely to be separable; feature $\mathbf{a}$ has a medium value of minimum mutual information of about 0.37 bits.

learning rate 0.1. With the solution $(\mathbf{v}^*, c^*)$, the final value of $M_{\epsilon, T=0}(\mathbf{f}, \mathbf{v}^*, c^*)$ is then our estimate of $M_\epsilon(\mathbf{f})$.

We also run the irreducibility tests on additional synthetic feature distributions in fig. 9.11a and fig. 9.11b.

9.7.3 Alternative Definitions

In this section, we present an alternative definition of a reducible feature that we considered during our work. This chiefly deals with multi-dimensional features from the angle of *computational* reducibility as opposed to *statistical* reducibility. In other words, this definition considers whether representations of features on a specific set of tasks can be split up without changing the accuracy of the task. This captures an interesting (and important) aspect of feature reducibility, but because it requires a specific set of prompts (as opposed to allowing unsupervised discovery) we chose not to use it as our main definition.

Our alternative definitions consider *representation spaces* that are possibly multi-dimensional, and defines these spaces through whether they can completely explain a function h on the

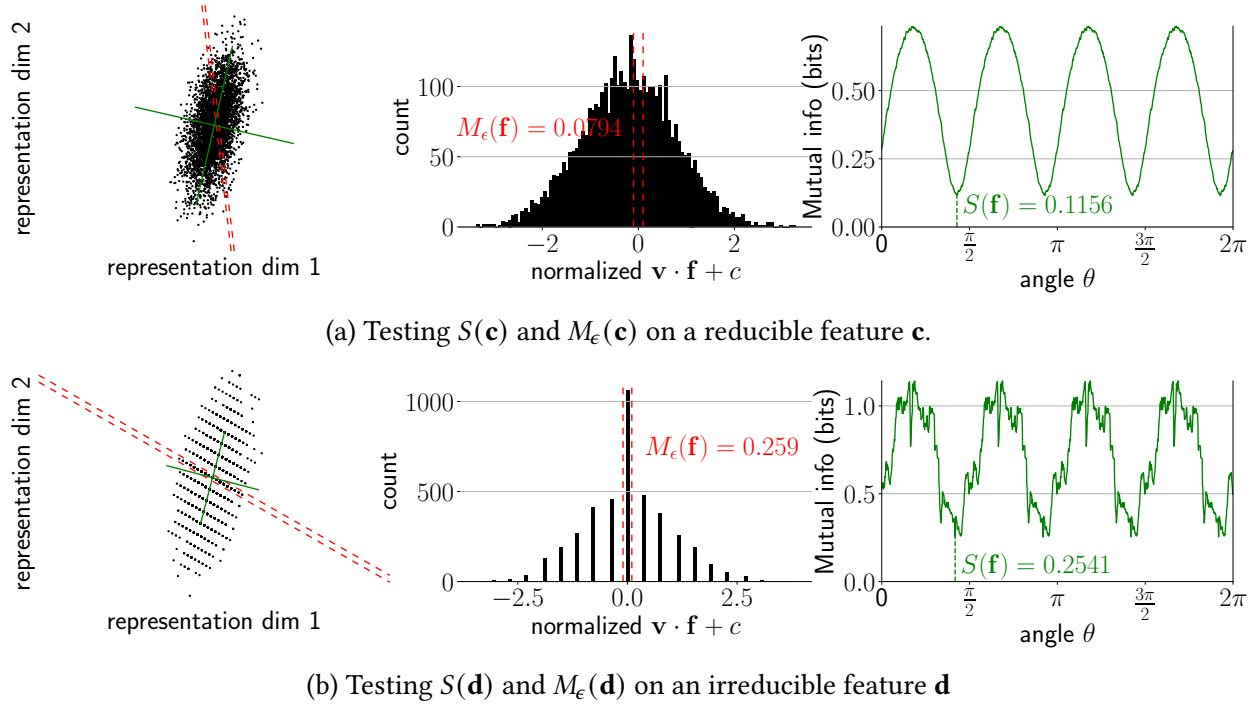


Figure 9.11: Testing irreducibility of synthetic features. **Left in each subfigure:** Distributions of $\mathbf{x}$. For feature $\mathbf{c}$, 7.94% lies within the narrow dotted lines, indicating the feature is unlikely to be a mixture. For feature $\mathbf{d}$, 25.90% lies within the wide lines, indicating the feature is likely a mixture. The green cross indicates the angle θ that minimizes mutual information. **Middle 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:** Mutual information between $\mathbf{a}$ and $\mathbf{b}$ as a function of the rotation angle θ of matrix $\mathbf{R}$. Both features have a small (< 0.5 bits) minimum mutual information and so are likely separable.

output logits. We consider a *group theoretic* approach to irreducible representations, via whether computation involving multiple group elements can be decomposed.

Alternative Definition: Interventions and Representation Spaces

Assume that we restrict the input set of prompts $T = \{\mathbf{t}^j\}$ to some subset of prompts and that we have some evaluation function h that maps from the output logit distribution of M to a real number. For example, for the Weekdays problems, T is the set of 49 prompts and h could be the argmax over the days of week logits. Abusing notation, we let M also be the function from the layer we are intervening on; this is always clear from context. Then we can define a representation space of $\mathbf{x}_{i,l}^j$ as a subspace in which interventions always work:

Definition 6 (Representation Space). Given a prompt set $T = \{\mathbf{t}^j\}$, a rank- r dimensional *representation space* of intermediate value $\mathbf{x}_{i,l}^j$ is a rank r projection matrix P such that for all j, j' , $h(M((I - P)\mathbf{x}_{i,l}^j + P\mathbf{x}_{i,l}^{j'})) = h(M(\mathbf{x}_{i,l}^{j'}))$.

Note that it immediately follows that the rank d dimensional matrix I_d is trivially a rank d representation space for all prompt sets T .

Definition 7 (Minimality). A representation space P of rank r is *minimal* if there does not exist a lower rank representation space.

A minimal representation with rank > 1 is a *multi-dimensional representation*.

Definition 8 (Alternative Reducibility). A representation space P of rank r is *reducible* if there are orthonormal representation spaces P_1 and P_2 (such that $P_1 + P_2 = P$, $P_1 P_2 = 0$) where

$$h(M(P_1 \mathbf{x}_{i,l}^j) + M(P_2 \mathbf{x}_{i,l}^j)) = h(M(P_1 \mathbf{x}_{i,l}^j + P_2 \mathbf{x}_{i,l}^j))$$

for all j, j' .

Suppose T , h and M define the multiplication of two elements in a finite group G of order n . Then if we interpret the embedding vectors as the group representations, our definition of reducibility implies to the standard group-theoretical definition of irreducibility -- specifically, reducibility into a tensor product representation.

9.7.4 Toy Case of Training SAEs on Circles

To explore how SAEs behave when reconstructing irreducible features of dimension $d_f > 1$, we perform experiments with the following toy setup. Inspired by the circular representations of integers that networks learn when trained on modular addition [7,8], we create synthetic datasets of activations containing multiple features which are each 2d irreducible circles.

First however, consider activations for a single circle – points uniformly distributed on the unit circle in $\mathbb{R}^2$. We train SAEs on this data with encoder $\text{Enc}(\mathbf{x}) = \text{ReLU}(\mathbf{W}_e(\mathbf{x} - \mathbf{b}_d) + \mathbf{b}_e)$ and decoder $\text{Dec}(\mathbf{f}) = \mathbf{W}_d \mathbf{f} + \mathbf{b}_d$. We train SAEs with $m = 2$ and $m = 10$ with the Adam optimizer and a learning rate of 10^{-3} , sparsity penalty $\lambda = 0.1$, for 20,000 steps, and a warmup of 1000 steps. In fig. 9.12 we show the dictionary elements of these SAEs. When $m = 2$, the SAE must use both

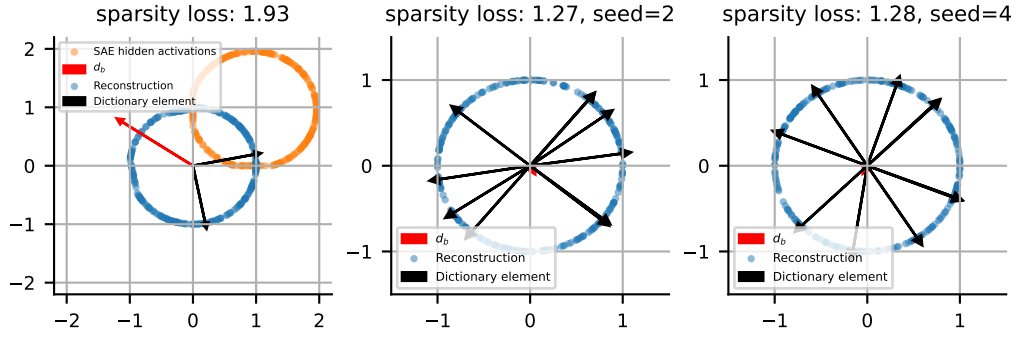


Figure 9.12: SAEs trained to reconstruct a single 2d circle with $m = 2$ (left) and $m = 10$ (middle and right) dictionary elements. When there are several SAE features, there is no natural/canonical choice of feature directions, and the dictionary elements spread out across the circle.

SAE features on each input point, and uses $\mathbf{d}_b$ to shift the reconstructed circle so it is centered at the origin. When $m = 10$, the SAE learns $\mathbf{d}_b \approx 0$ and the features spread out across the circle, arranged close together, and only a subset are active on each input.

We now consider synthetic activations with multiple circular features. Our data consists of points in $\mathbb{R}^{10}$, where we choose two orthogonal planes spanned by $(\mathbf{e}_1, \mathbf{e}_2)$ and $(\mathbf{e}_3, \mathbf{e}_4)$, respectively. With probability one half a points is sampled uniformly on the unit circle in the $\mathbf{e}_1$ - $\mathbf{e}_2$ plane, otherwise the point will be sampled uniformly on the unit circle in the $\mathbf{e}_3$ - $\mathbf{e}_4$ plane. We train SAEs with $m = 64$ on this data with the same hyperparameters as the single-circle case.

We now apply the procedure described in section 9.4 to see if we can automatically rediscover these circles. Encouragingly, we first find that the alive SAE features align almost exactly with either the $\mathbf{e}_1$ - $\mathbf{e}_2$ or the $\mathbf{e}_3$ - $\mathbf{e}_4$ plane. When we apply spectral clustering with `n_clusters = 2` to the features with the pairwise angular similarities between dictionary elements as the similarity matrix (fig. 9.13, left), the two clusters correspond exactly to the features which span each plane. As described in section 9.4, given a cluster of dictionary elements $S \subset \{1, \dots, m\}$, we run a large set of activations through the SAE, then filter out samples which don't activate any element in S . For samples which do activate an element of S , reconstruct the activation while setting all SAE features not in S to have a hidden activation of zero. If some collection of SAE features together represent some irreducible feature, we want to remove all other features from the activation vector, and so we only allow SAE features in the collection to participate in reconstructing the input activation. We find that this procedure almost exactly recovers the original two circles, which encouraged us to apply this method for discovering the features shown in fig. 9.1 and fig. 9.15.

9.7.5 Training Mistral SAEs

Our Mistral 7B [40] sparse autoencoders (SAEs) are trained on over one billion tokens from a subset of the Pile [54] and Alpaca [55] datasets. We train our SAEs on layers 8, 16, and 24 out of 32 total layers to maximize coverage of the model's representations. We use a $16\times$ expansion factor, yielding a total of 65536 dictionary elements for each SAE.

To train our SAEs, we use an L_p sparsity penalty for $p = 1/2$ with sparsity coefficient $\lambda = 0.012$. Before an SAE forward pass, we normalize our activation vectors to have norm $\sqrt{d_{\text{model}}} = 64$ in

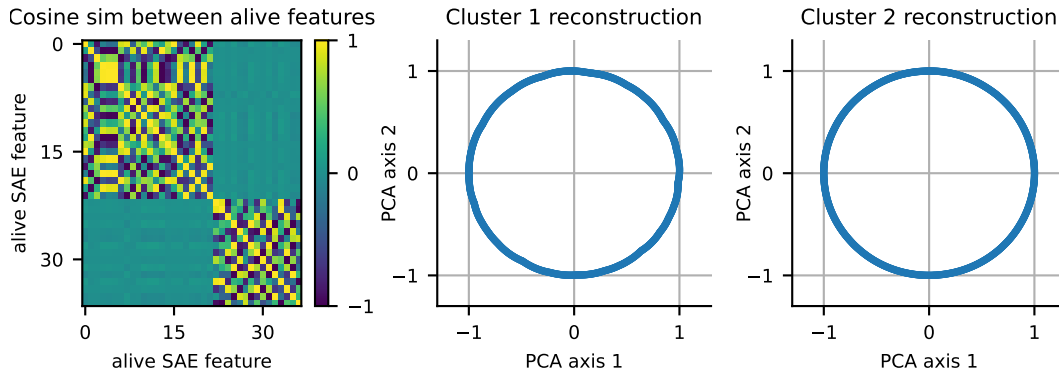


Figure 9.13: Automatic discovery of synthetic circular features by clustering SAE dictionary elements.

the case of Mistral. We do not apply a pre-encoder bias. We use an AdamW optimizer with weight decay 10^{-3} and learning rate 0.0002 with a linear warm up. We apply dead feature resampling [12] five times over the course of training to converge on SAEs with around 1000 dead features.

9.7.6 GPT-2 and Mistral 7B Dictionary Element Clustering

In this section, we first present pseudocode in line 1 for the overall high level technique that finds multi-dimensional features and that uses clustering as a subroutine. We then provide the specific clustering algorithm implementations we use for GPT-2 and Mistral.

Algorithm 1: High Level Clustering Approach For Finding Multi-D Features

Input: Dictionary elements D , activation vectors $X_{i,l}$, SAE

Output: Irreducible multi-dimensional features

$S_{i,j} \leftarrow \text{CosineSim}(D_i, D_j);$

$clusters \leftarrow \text{Cluster}(S);$

$reconstructions \leftarrow \{\};$

for $cluster$ **in** $clusters$ **do**

$R_{cluster} \leftarrow$ ids of dictionary elements in $cluster$;

for $x_{i,l}$ **in** $X_{i,l}$ **do**

$encoding \leftarrow \text{ReLU}(E \cdot x_{i,l});$

if $\max(encoding[R_{cluster}]) > 0$ **then**

$r \leftarrow D[:, R_{cluster}] \cdot encoding;$

$reconstructions \leftarrow reconstructions \cup \{r\};$

end

end

end

$features \leftarrow \{\};$

for R **in** $reconstructions$ **do**

$proj \leftarrow \text{PCA}(R);$

if $\text{TestIrreducible}(proj)$ **then**

 Add $proj$ to $features$;

end

end

return $features$

GPT-2-small methods and results

For GPT-2-small, we perform spectral clustering on the roughly 25k layer 7 SAE features from [39], using pairwise angular similarities between dictionary elements as the similarity matrix. We use `n_clusters = 1000` and manually looked at roughly 500 of these clusters. For each cluster, we looked at projections onto principal components 1-4 of the reconstructed activations for these clusters. In fig. 9.14, we show projections for the most interesting clusters we identified, which appear to be circular representations of days of the week, months of the year, and years of the 20th century.

Mistral 7B methods and results

For Mistral 7B, our SAEs have 65536 dictionary elements and we found it difficult to run spectral clustering on all of these at once. We therefore develop a simple graph based clustering algorithm that we run on Mistral 7B SAEs:

1. Create a graph G out of the dictionary elements by adding directed edges from each dictionary element to its k closest dictionary elements by cosine similarity. We use $k = 2$.
2. Make the graph undirected by turning every directed edge into an undirected edge.

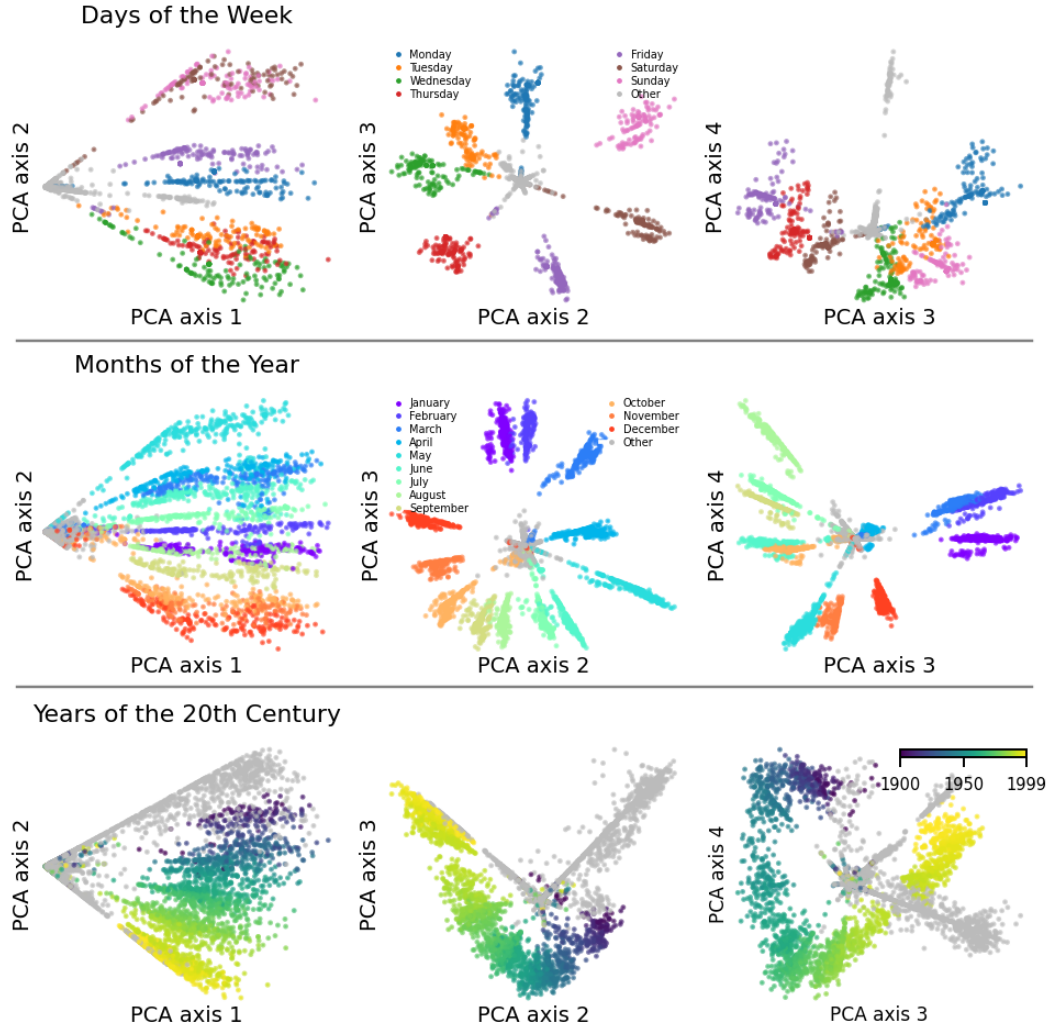


Figure 9.14: Projections of days of week, months of year, and years of the 20th century representations onto top four principal components, showing additional dimensions of the representations than fig. 9.1.

3. Prune edges with cosine similarity less than a threshold value τ . We use $\tau = 0.5$.
4. Return the connected components as clusters.

We run this algorithm on the Mistral 7B layer 8 SAE (2^{16} dictionary elements) and find roughly 2700 clusters containing between 2 and 1000 elements. We manually inspected roughly 2000 of these. From these, we re-discover circular representations of days of the week and months of the year, shown in fig. 9.15. However, we did not find other obviously interesting and clearly irreducible features.

We also investigate the sensitivity of this method to τ and k by varying τ and k and showing the max Jaccard similarity between any of the resulting clusters and the days of the week cluster we show in fig. 9.15. We show the results in fig. 9.16, where we find that varying k has minimal effect, while varying τ shows 3 regimes: small τ causes all features to group in one cluster, so the

days of the week cluster is not found; medium τ causes the days of the week cluster to become identifiable; large τ causes all features to be divided into their own clusters.

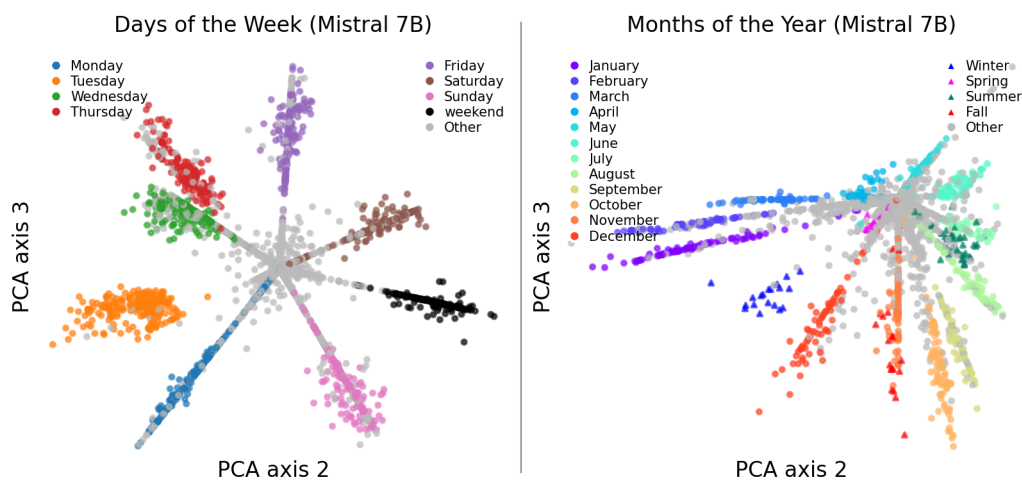


Figure 9.15: Circular representations of days of the week and months of the year which we discover with our unsupervised SAE clustering method in Mistral 7B. Unlike similar features in GPT-2, we also find an additional “weekend” representation in between Saturday and Sunday representations (left) and additional representations of seasons among the months (right). For instance, “winter” tokens activate a region of the circle in between the representation of January and December.

As future work, we think it would be exciting to develop better clustering techniques for SAE features. Our graph based clustering technique could likely be improved by more recent efficient and high-quality graph based clustering techniques, e.g. hierarchical agglomerate clustering with single-linkage [56]. Additionally, we believe we would see a large improvement by setting edge weights to be a combination of both the cosine and Jaccard similarity of the dictionary elements, e.g. $\max(\text{cosine}, \text{Jaccard})$.

9.7.7 Other discovered clusters

In fig. 9.17, we plot the top 11 ranked clusters by the product of a) the measured separability index and b) one minus the measured ϵ -mixture index with $\epsilon = 0.1$ (this is just one of many possible ways to get an ordered ranking from a two-parameter score). We color by both the current token (which results in clear patterns for all tokens) and the next token (to see if we find belief states as found by [28] in toy transformers). We note that weekdays are ranked 9 and so are shown in the plot. Additionally, the next token patterns of the ‘such’ cluster and the ‘B’ cluster do seem to display some clustering independently of the the current token pattern, which might lend the belief state hypothesis some support.

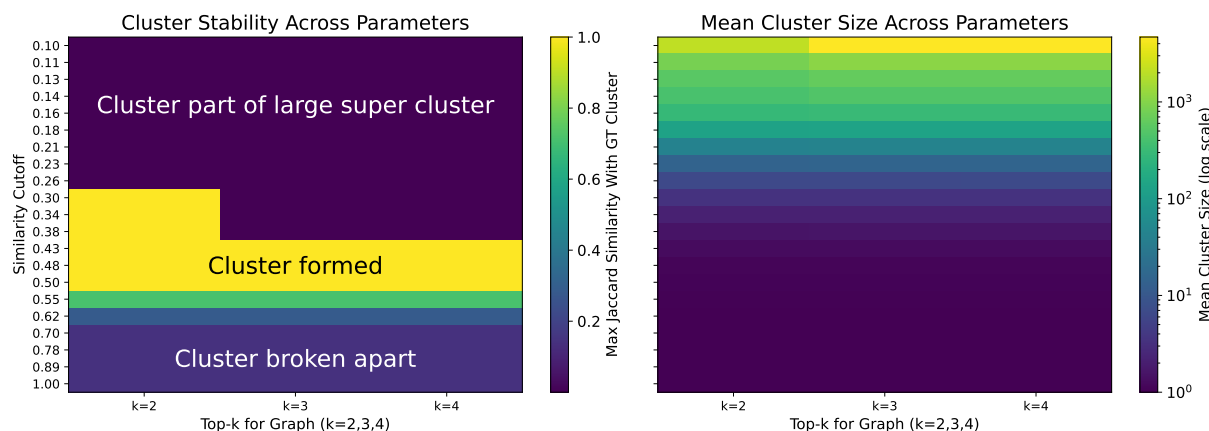


Figure 9.16: Hyperparameter regimes where the days of the week cluster exists. The cluster exists in the regime between all features clumping together and all features being in their own cluster; this regime seems reasonably stable.

9.7.8 Further Experiment Details

Assets Information

We use the following open source models for our experiments: Llama 3 8B [41] (custom Llama 3 license <https://llama.meta.com/llama3/license/>), Mistral 7B [40] (released under the Apache 2 License), and GPT-2 [38] (modified MIT license, see <https://github.com/openai/gpt-2/blob/master/LICENSE>).

Machine Information

Intervention experiments were run on two V100 GPUs using less than 64 GB of CPU RAM; all experiments can be reproduced from our open source repository in less than a day with this configuration. We use the TransformerLens library [57] for intervention experiments. ϵ -mixture index measurements on toy datasets took about one minute each, on 8GB of CPU RAM. EVR experiments take seconds on 8GB of CPU RAM and are dominated by time taken to human-interpret the RGB plots.

GPT-2 SAE clustering and plotting was run on a cluster of heterogeneous hardware. Spectral clustering and computing reconstructions + plotting was done on CPUs only. We made reconstruction plots for 500 clusters, with each taking less than 10 minutes. Mistral 7B SAE reconstruction plots were made on the same cluster. We made roughly 2000 reconstruction plots for Mistral 7B (and manually inspected each), with each taking less than 20 minutes to generate. Jobs were allocated 64GB of memory each.

Mistral SAE training was run on a single V100 GPU. Initially caching activations from Mistral 7B on one billion tokens took approximately 60 hours. Training the SAEs on the saved activations took another 36 hours.

Error Bar Calculation

In fig. 9.6 we report 96% error bars for all intervention methods. To compute these error bars, we loop over all intervention methods and all layers and compute a confidence interval for each (method, layer) pair across all prompts. Assuming normally distributed errors, we compute error bars with the following standard formula:

$$EB = \mu \pm z * SE$$

where μ is the sample mean, z is the z score (slightly larger than 2 for 96% error bars), and SE is the standard error (the standard deviation divided by the square root of the number of samples). We use standard Python functions to compute this value.

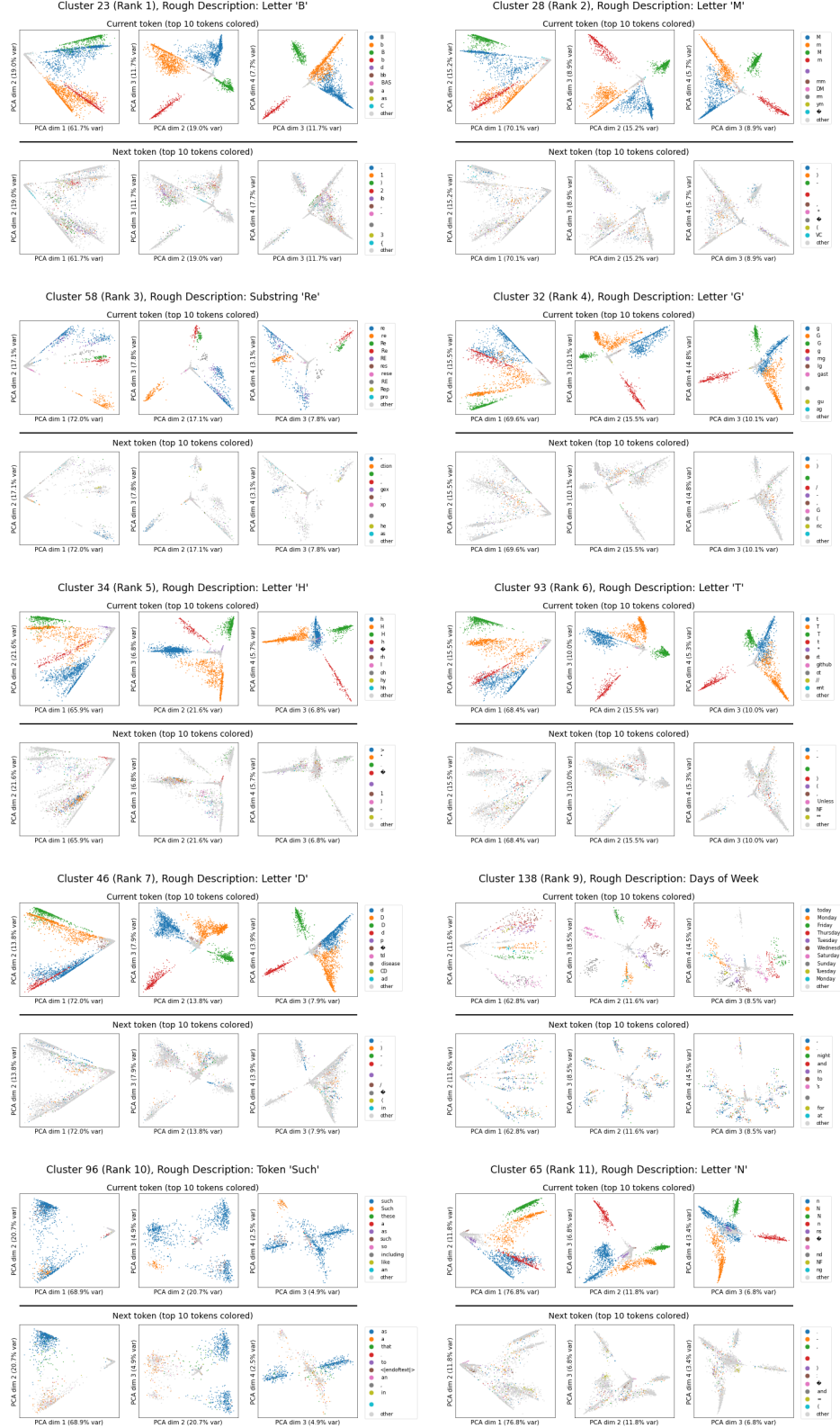


Figure 9.17: Top 10 GPT-2 clusters by Mixture and Separability Index.

The reason that the Months error bars are smaller than the Weekdays error bars is because

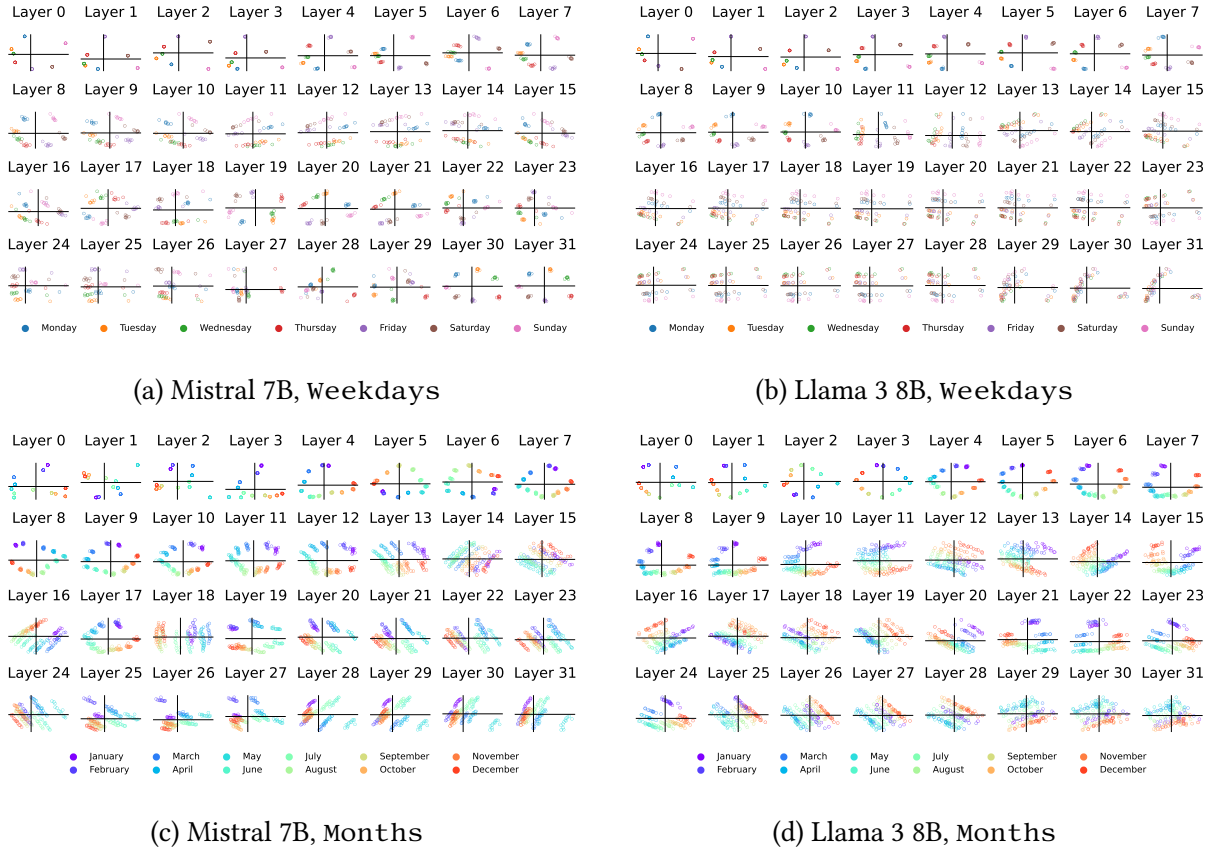


Figure 9.18: Projections onto the top two PCA dimensions of model hidden states on the α token show that circular representations of α are present in various layers.

there are more Months prompts: there are $12 * 12 * 11 = 1584$ intervention effect values, rather than $7 * 7 * 6 = 294$ intervention effect values.

9.7.9 More Weekdays and Months Plots and Details

Basic Plots

We show the results of Mistral 7B and Llama 3 8B on all individual instances of Weekdays that at least one of the models get wrong in table 9.2 and present a similar table for Months in table 9.3.

We show projections onto the top two PCA directions for both Mistral 7B and Llama 3 8B in fig. 9.18 on the hidden layers on top of the α token, colored by α . These are similar plots to fig. 9.4, except they are on all layers. The circular structure in α is visible on many—but not all—layers. Much of the linear structure visible is due to β .

Intervening with the SAE Probe

We show the results of projecting Mistral Weekdays representations into the plane discovered by clustering SAE features in fig. 9.21. The result is clearly circular.

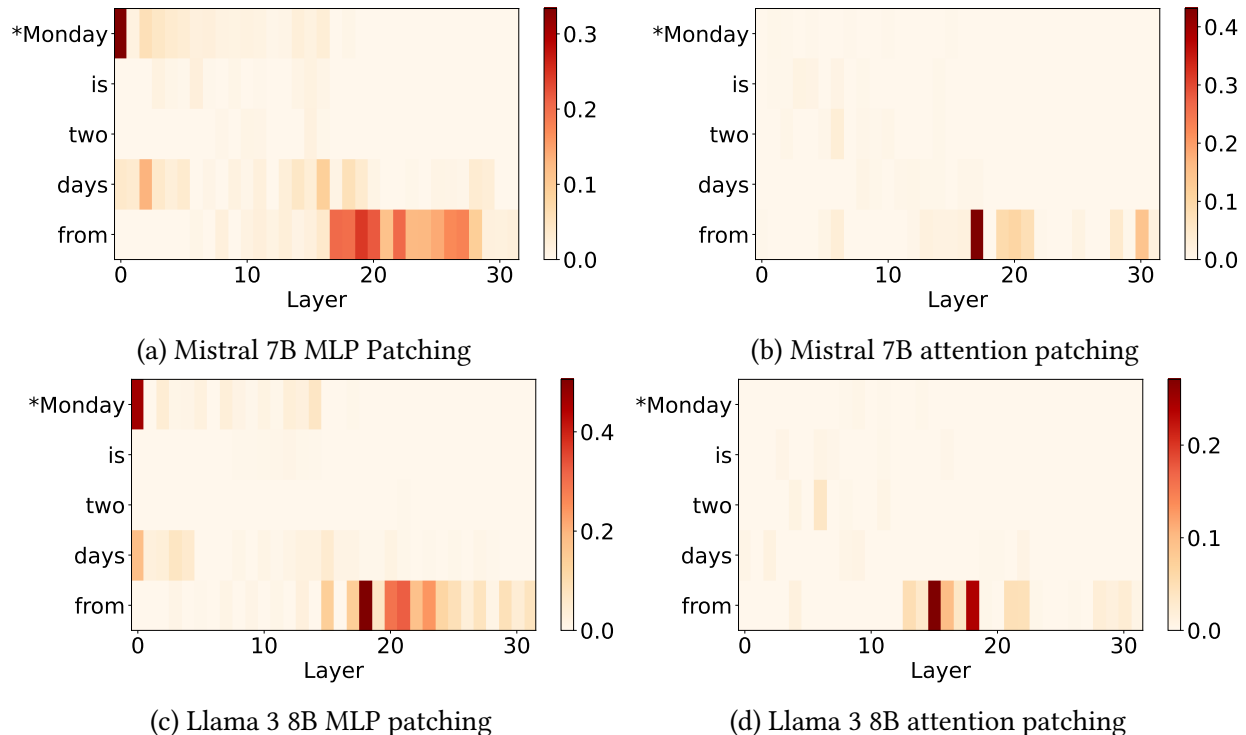


Figure 9.19: Attention and MLP patching results on Weekdays. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α .

Basic Patching

In fig. 9.19 and fig. 9.20, we report MLP and attention head patching results for Weekdays and Months. We experiment on 20 pairs of problems with the same α and different β and 20 pairs of problems with the same β and different α , for a total of 40 pairs of problems. For each pair of problems, we patch the MLP/attention outputs from the "clean" to the "dirty" problem for each layer and token, and then complete the forward pass. Defining the logit difference as the logit of the clean y minus the logit of the dirty y , we record what percent of the difference between the original logit difference of the dirty problem and the logit difference of the clean problem is recovered upon intervening, and average across these 40 percent-ages for each layer and token. This gives us a score we call the *Average Intervention Effect*.

For simplicity of presentation, we clip all of the (few) negative intervention averages to 0 (prior work [42] has also found negative-effect attention heads during patching experiments).

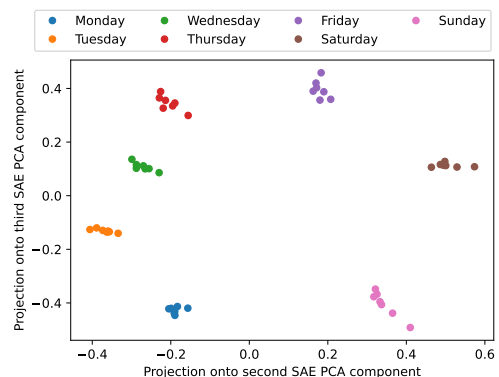


Figure 9.21: Projections of Mistral 7B Weekdays task activations at layer 8 into the plane discovered by clustering layer 8 SAE features.

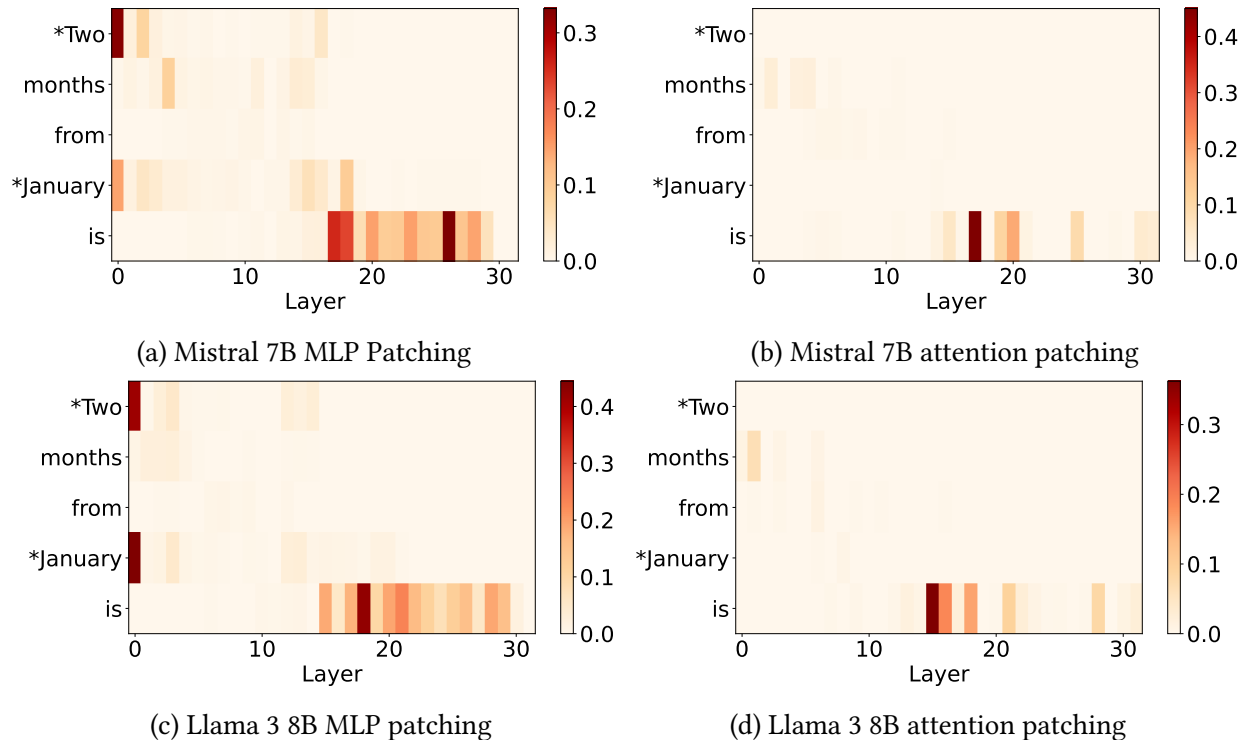


Figure 9.20: Attention and MLP patching results on Months. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α .

Circle Continuity

Finally, in fig. 9.22, we show another example of the continuity of the circular days of the week representation in Mistral 7B.

9.7.10 Patching

In this section, we present results to support a claim that MLPs (and not attention blocks) are responsible for computing γ . In fig. 9.25, we deconstruct states on top of the final token (before predicting γ) on Llama 3 8B Months (we show a similar plot for the states on the final token of Mistral 7B on Weekdays in the main text in fig. 9.24). These plots show that the value of γ is computed on the final token around layers 20 to 25. To show that this computation of occurs in the MLPs, we must show that no attention head is copying γ from a prior token or directly computing γ .

We first perform a patching experiment with the same setup fig. 9.20 and fig. 9.19 on individual attention heads on the final token. From the patching results we identify

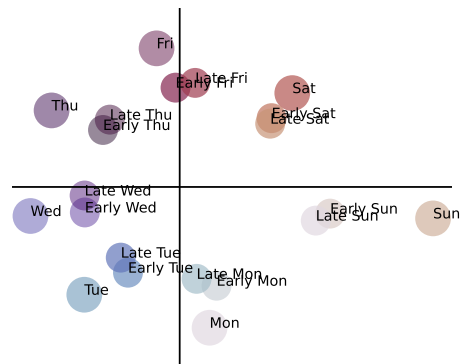


Figure 9.22: Layer 30 Mistral 7B activations for [very early/very late] on [Monday/Tuesday/.../Sunday], plotted into the PCA plane for [Monday/Tuesday/.../Sunday].

Table 9.2: Weekdays finegrained results. Row ommited if both models get it correct.

α	β	Ground truth γ	Mistral top γ	Mistral correct?	Llama top γ	Llama correct?
1	1	Wednesday	Wednesday	Yes	Thursday	No
3	1	Friday	Friday	Yes	Tuesday	No
4	1	Saturday	Saturday	Yes	Thursday	No
3	2	Saturday	Saturday	Yes	Tuesday	No
4	2	Sunday	Sunday	Yes	Wednesday	No
5	2	Monday	Monday	Yes	Tuesday	No
2	3	Saturday	Friday	No	Saturday	Yes
3	3	Sunday	Sunday	Yes	Tuesday	No
4	3	Monday	Monday	Yes	Tuesday	No
0	4	Friday	Thursday	No	Friday	Yes
3	4	Monday	Monday	Yes	Tuesday	No
0	5	Saturday	Friday	No	Saturday	Yes
1	5	Sunday	Saturday	No	Wednesday	No
2	5	Monday	Sunday	No	Monday	Yes
4	5	Wednesday	Tuesday	No	Tuesday	No
6	5	Friday	Thursday	No	Thursday	No
1	6	Monday	Sunday	No	Thursday	No
2	6	Tuesday	Monday	No	Tuesday	Yes
3	6	Wednesday	Tuesday	No	Tuesday	No
4	6	Thursday	Thursday	Yes	Tuesday	No
5	6	Friday	Friday	Yes	Thursday	No
6	6	Saturday	Thursday	No	Thursday	No
0	7	Monday	Sunday	No	Tuesday	No
1	7	Tuesday	Sunday	No	Tuesday	Yes
2	7	Wednesday	Sunday	No	Wednesday	Yes
3	7	Thursday	Sunday	No	Thursday	Yes
4	7	Friday	Thursday	No	Tuesday	No
5	7	Saturday	Friday	No	Saturday	Yes
6	7	Sunday	Friday	No	Thursday	No

Table 9.3: Months finegrained results. Row omitted if both models get it correct.

α	β	Ground truth γ	Mistral top γ	Mistral correct?	Llama top γ	Llama correct?
0	4	May	April	No	May	Yes
6	4	November	October	No	November	Yes
0	6	July	June	No	July	Yes
0	7	August	July	No	August	Yes
1	7	September	October	No	September	Yes
3	7	November	October	No	November	Yes
5	7	January	December	No	January	Yes
6	7	February	January	No	February	Yes
7	7	March	February	No	March	Yes
9	7	May	April	No	May	Yes
4	9	February	February	Yes	January	No
2	10	January	December	No	January	Yes
8	10	July	June	No	July	Yes
1	11	January	December	No	January	Yes
2	11	February	December	No	February	Yes
3	11	March	February	No	March	Yes
7	11	July	June	No	July	Yes
8	11	August	July	No	August	Yes
9	11	September	August	No	September	Yes
0	12	January	December	No	January	Yes

the top 10 attention heads by average intervention effect.

For each attention head, we compute one EVR run with explanatory functions equal to one-hot functions of α

and β (resulting in 14 functions $\mathbf{g}_i$ for Weekdays and 24 for Months) and one with explanatory functions equal to one-hot functions of α , β , and γ . We find that for all layers before 25, adding γ to the explanatory functions adds almost no explanatory power. Since we established above that the model has already computed γ at this point, we know that attention heads do not participate in computing γ .

To isolate the rough circuit for Weekdays and Months, we perform layer-wise activation patching on 40 random pairs of prompts. The results, displayed in fig. 9.20 show that the circuit to compute γ consists of MLPs on top of the α and β tokens, a copy to the token before γ , and further MLPs there (roughly similar to prior work studying arithmetic circuits [37]). Moreover, fine-grained patching in section 9.7.11 shows that there are just a few responsible attention heads for the writes to the token before γ . However, patching alone cannot tell use *how* or *where* γ is represented. For that, we need a new technique, which we expand on in the next section.

9.7.11 Explanation via Regression (EVR)

So far, we have focused on examining and intervening on the representation for α , which we present as a circle in the top PCA components on top of the α token. In this section, we examine how the generated output, γ , is represented.

First, to isolate the rough circuit for Weekdays and Months, we perform layer-wise activation patching on 40 random pairs of prompts. The results, displayed in fig. 9.20 and fig. 9.19, show that the circuit to compute γ consists of MLPs on top of the α and β tokens, a copy to the token before γ , and further MLPs there (roughly similar to what [37] find in prior work studying arithmetic circuits). Thus, we know *where* to look for a representation of γ : in the second half of the layers on the token before γ . However, patching alone cannot tell use *how* γ is represented.

Unlike α , γ has no obvious circular (or linear) pattern in the top PCA components on these layers. To determine the representation for γ , we introduce a more powerful technique we call Explanation via Regression (EVR): given a set of token sequences with a corresponding set of hidden states $X_{i,l}$, we choose a set of interpretable explanation functions of the input tokens $\{\mathbf{g}_j(t)\}$. The r^2 value of a linear regression from $\{\mathbf{g}_j(t)\}$ to $X_{i,l}$ tells us how much of the variance in the activations the $\{\mathbf{g}_j(t)\}$ explain, and conversely the residuals show the exact components of the representation we have yet to explain.

Using EVR to uncover a circular representation for γ

We first use EVR to determine the representation for γ by plotting the top two PCA components of the layer 25 Mistral 7B activations after subtracting the components that can be explained using a regression with one hot functions in α and β (i.e. $\mathbf{g}_1 = [\alpha = 0]$, $\mathbf{g}_2 = [\beta = 1]$, $\mathbf{g}_3 = [\alpha = 1], \dots$).

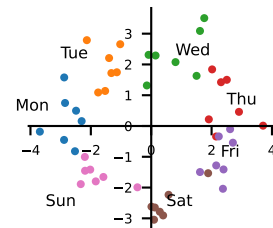


Figure 9.23: Top two PCA components of residual errors after EVR with one-hot in α and β . Mistral 7B Weekdays, layer 25, final token. Colored by γ .

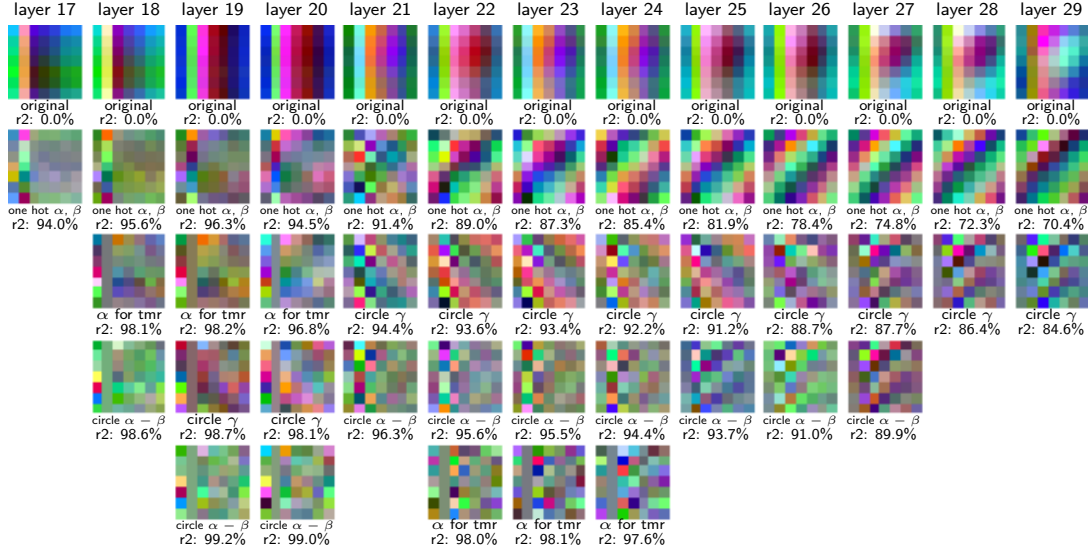


Figure 9.24: EVR residual RGB plots on Mistral hidden states on the Weekdays final token, layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) $\mathbf{g}_i$ labelled just underneath, as well as the resulting r^2 value. We write “tmr” meaning “tomorrow” for $\beta = 1$. We also write “circle for x ” meaning the inclusion of two functions $\mathbf{g}_i(x) = \{\cos, \sin\}(2\pi x/7)$.

The result, shown in fig. 9.23, is an incredibly clear circle in γ , which suggests that the model’s generated representation of γ lies along a circle. A simple PCA projection was not enough to find this result because the representation for γ has interference from α and β , which the EVR removes. This suggests that the models may be generating γ by using a trigonometry based algorithm like the “clock” [8] or “pizza” [36] algorithm in late MLP layers.

More Experiments with EVR

We now apply EVR to Months and Weekdays to break down $X_{i,l}$ completely into interpretable functions. We build a list of $\mathbf{g}_i$ *iteratively* and *greedily*. At each iteration, we perform a linear regression with the current list $\mathbf{g}_1 \dots \mathbf{g}_k$, visualize and interpret the residual prediction errors, and build a new function $\mathbf{g}_{k+1}$ representing these errors to add to the list. Once most variance is explained, we can conclude that $\mathbf{g}_1, \dots, \mathbf{g}_k$ constitutes the entirety of what is represented in the hidden states. This information tells us what can and cannot be extracted via a linear probe, without having to train any probes. Furthermore, if we treat each $\mathbf{g}_i$ as a *feature* (see definition 2), then the linear regression coefficients tell us which directions in $X_{i,l}$ these features are represented in, connecting back to Hypothesis 2.

Since $X_{i,l}$ consists of modular addition problems with two inputs α and β , we can visualize the errors as we iteratively construct $\mathbf{g}_1, \dots, \mathbf{g}_k$ 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.

In fig. 9.24, we perform EVR on the layer 17-29 hidden states of Mistral 7B on the Weekdays task; additional deconstructions are in section 9.7.11. We find that a circle in γ develops and grows in explanatory power; we plot the layer 25 residuals after explaining with one hot functions in α and β (i.e. $\mathbf{g}_1 = [\alpha = 0]$, $\mathbf{g}_2 = [\beta = 1]$, $\mathbf{g}_3 = [\alpha = 1], \dots$) in fig. 9.23 to show this incredibly clear circle in γ . This suggests that the models may be generating γ by using a trigonometry based algorithm like the “clock” [8] or “pizza” [36] algorithm in late MLP layers.

Table 9.4: Highest intervention effect attention heads from fine-grained attention head patching, as well as EVR results with one hot α , β and one hot α , β , γ .

(a) Mistral 7B, Weekdays.					(b) Llama 3 8B, Weekdays.				
L	H	Average Inter- ven- tion Effect	EVR R^2 One Hot α , β	EVR R^2 One Hot α , β, γ	L	H	Average Inter- ven- tion Effect	EVR R^2 One Hot α , β	EVR R^2 One Hot α , β, γ
28	18	0.22	0.39	0.73	17	0	0.18	0.98	0.99
18	30	0.17	0.95	0.96	17	1	0.08	0.98	0.98
15	13	0.17	0.94	0.95	19	10	0.08	0.95	0.96
22	15	0.11	0.77	0.82	30	17	0.07	0.85	0.90
16	21	0.09	0.92	0.93	17	3	0.07	0.93	0.95
28	16	0.08	0.42	0.69	17	27	0.06	1.00	1.00
15	14	0.06	0.98	0.99	31	22	0.05	0.37	0.78
30	24	0.05	0.43	0.79	21	9	0.04	0.73	0.78
21	26	0.04	0.53	0.63	20	28	0.04	1.00	1.00
14	2	0.04	0.93	0.95	30	16	0.04	0.73	0.85
(c) Mistral 7B, Months.					(d) Llama 3 8B, Months.				
L	H	Average Inter- ven- tion Effect	EVR R^2 One Hot α , β	EVR R^2 One Hot α , β, γ	L	H	Average Inter- ven- tion Effect	EVR R^2 One Hot α , β	EVR R^2 One Hot α , β, γ
20	28	0.15	0.76	0.76	15	13	0.26	0.62	0.62
17	0	0.10	0.77	0.77	16	21	0.17	0.76	0.76
25	14	0.08	0.19	0.61	18	30	0.13	0.77	0.77
17	1	0.07	0.80	0.82	28	18	0.11	0.13	0.52
17	3	0.06	0.71	0.71	28	16	0.07	0.13	0.52
31	22	0.06	0.12	0.67	21	25	0.05	0.65	0.70
17	27	0.05	0.58	0.58	15	14	0.03	0.72	0.72
19	4	0.05	0.40	0.66	17	26	0.02	0.77	0.77
19	10	0.04	0.62	0.62	31	1	0.02	0.11	0.57
30	26	0.04	0.51	0.62	21	24	0.02	0.30	0.45

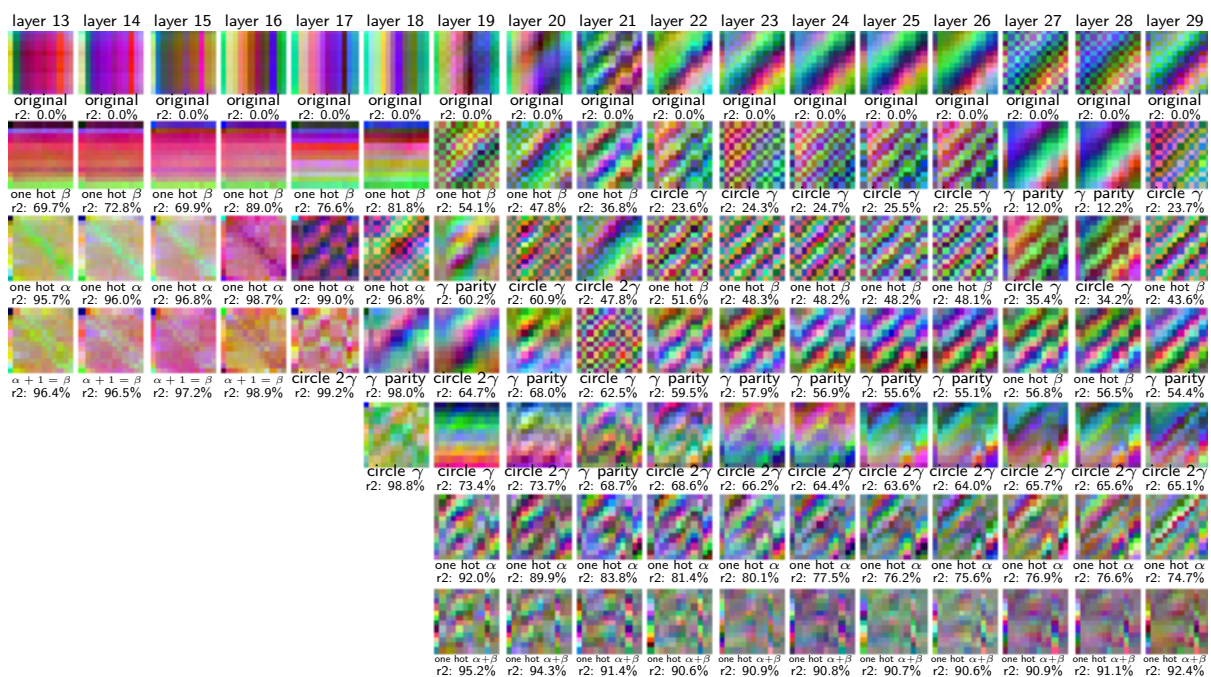


Figure 9.25: Iterative deconstruction of hidden state representations on the final token on Llama 3 8B, Months.

Chapter 10

Understanding Sparse Autoencoder Scaling in the Presence of Feature Manifolds^{*}

This final chapter ties together many threads from throughout this thesis: sparse autoencoders, scaling laws, and feature geometry. It develops a model of scaling laws for sparse autoencoders, adapting our model of neural scaling from Chapter 2 (or rather a model from the literature which generalized our model) to study sparse autoencoder scaling. It is particularly interested in how feature manifolds, which we showed existed in large language models in Chapter 9, impact SAE scaling.

Abstract

Sparse autoencoders (SAEs) model the activations of a neural network as linear combinations of sparsely occurring directions of variation (latents). The ability of SAEs to reconstruct activations follows scaling laws w.r.t. the number of latents. In this work, we adapt a capacity-allocation model from the neural scaling literature (Brill, 2024) to understand SAE scaling, and in particular, to understand how *feature manifolds* (multi-dimensional features) influence scaling behavior. Consistent with prior work, the model recovers distinct scaling regimes. Notably, in one regime, feature manifolds have the pathological effect of causing SAEs to learn far fewer features in data than there are latents in the SAE. We provide some preliminary discussion on whether or not SAEs are in this pathological regime in the wild.

10.1 Introduction

Sparse autoencoders [1–9] and related methods [10–14] decompose neural network activations into a collection of sparsely activating latents. As SAEs have been scaled (now to millions of latents) [4,15–17], they have exhibited *scaling laws* [4,15,18], where loss improves predictably as a power law with the number of latents in the SAE. Although sparse autoencoders learn many

^{*}Eric J. Michaud, Liv Gorton, and Tom McGrath. “Understanding sparse autoencoder scaling in the presence of feature manifolds.” *Mechanistic Interpretability Workshop at NeurIPS* (2025). Preprint: [arXiv:2509.02565](https://arxiv.org/abs/2509.02565).

interpretable features, they may miss important structure in neural representations, either because there are an extraordinary number of very rare features in activations [19] or because the SAE architecture and training objective make incorrect assumptions about the structure of neural representations [20,21].

In this work, we develop a formal analysis of SAE scaling behavior and scaling laws. We are particularly interested in understanding SAE scaling when activations contain a particular kind of structure: *feature manifolds* (multi-dimensional features) [22,23], and in whether feature manifolds could cause pathological scaling and exacerbate the problem of interpretability “dark matter” [19,24]. Our main approach is to adapt a mathematical model of *neural scaling* from Brill (2024) [25], where models allocate capacity between different data manifolds, to the case of SAEs. Guided by our model, we then conduct experiments to probe whether SAEs may scale pathologically in practice.

10.2 A model of sparse autoencoder scaling

10.2.1 The structure of data and the SAE architecture

Activation data: We assume the multi-dimensional linear representation hypothesis [24,26] and that neural network activation vectors $\mathbf{x} \in \mathbb{R}^d$ are generated as a sum of sparsely occurring *features*: $\mathbf{x} = \sum_i \mathbf{S}_i \mathbf{f}_i$ where $\mathbf{S}_i$ is the subspace where feature i lives, specified by a $d \times d_i$ matrix whose columns are basis vectors of the subspace, and $\mathbf{f}_i$ is a random variable taking values in $\mathbb{R}^{d_i}$, where d_i is the *dimension* of feature i . Each feature $\mathbf{f}_i$ is *sparse*, supported on a small fraction of the data: $p_i = \Pr[\mathbf{f}_i \neq \mathbf{0}] \ll 1$, so each activation vector is a sum of only a small subset of all features.

SAEs: Sparse autoencoders attempt to reconstruct activation vectors as a sum of sparsely activating latents, consisting of an encoding step $\hat{\mathbf{f}} = \text{Enc}(\mathbf{x}) = \sigma(\mathbf{W}_e \mathbf{x} + \mathbf{b}_e)$ where σ is a nonlinearity and $\hat{\mathbf{f}} \in \mathbb{R}^N$, and a decoding step $\hat{\mathbf{x}} = \text{Dec}(\hat{\mathbf{f}}) = \mathbf{W}_d \hat{\mathbf{f}} + \mathbf{b}_d$. Sparse autoencoders are trained with SGD on a loss $\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 + \lambda S(\hat{\mathbf{f}})$ where $S(\hat{\mathbf{f}})$ is a sparsity-encouraging loss like $\|\hat{\mathbf{f}}\|_1$ or $\|\hat{\mathbf{f}}\|_0$.

10.2.2 Assumption: SAE optimization reduces to a latent allocation problem

We assume that SAEs learn solutions where each latent j is *specific* to a particular feature i , so that $\hat{\mathbf{f}}_j \neq 0$ only if $\mathbf{f}_i \neq \mathbf{0}$. We further assume that the SAE latent decoder directions for the latents associated with feature i lie within the subspace $\mathbf{S}_i$, and that the subspaces where features live are orthogonal, i.e., $\mathbf{S}_i \perp \mathbf{S}_k$ if $i \neq k$. It follows that the SAE loss on any sample $\mathbf{x}$ is the sum of losses on samples where each active feature had fired alone: if $\mathcal{A}(\mathbf{x})$ are the “active” features i where $\mathbf{f}_i \neq \mathbf{0}$ on $\mathbf{x}$, then

$$\mathcal{L}(\mathbf{x}) = \mathcal{L}\left(\sum_{i \in \mathcal{A}(\mathbf{x})} \mathbf{S}_i \mathbf{f}_i\right) \stackrel{\text{assumption}}{=} \sum_{i \in \mathcal{A}(\mathbf{x})} \mathcal{L}(\mathbf{S}_i \mathbf{f}_i).$$

We note that feature absorption and related phenomena [27] and the non-orthogonality of features in practice violate these assumptions, but we will accept them for the sake of expedience. If SAE latents are specific to features, then an optimal SAE’s loss on a feature i will be determined by the number of latents n_i the SAE allocates to reconstructing feature i , which we denote $L_i(n_i)$, and

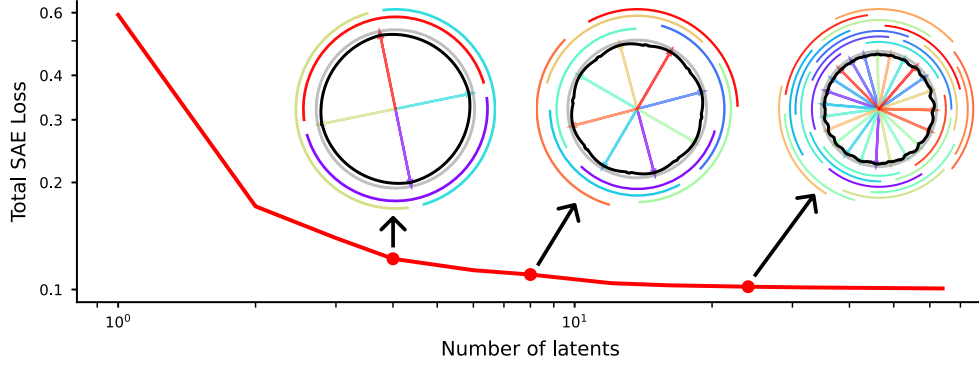


Figure 10.1: **SAE scaling on a toy feature manifold S^1 .** We train ReLU SAEs with an L1 sparsity penalty to reconstruct points on the circle $S^1 \subseteq \mathbb{R}^2$. We find that SAEs can slightly reduce their total loss by “tiling” the manifold with more sparsely activating latents. For SAEs with 4, 8, and 24 latents, we show the data (S^1) in grey, the SAE’s reconstruction in black, the decoder latent directions as arrows, and indicate the part of the circle that each latent fires on as colored arcs. If SAEs can reduce loss by “tiling” a manifold in this way, they may do this at the expense of learning rarer features.

will be determined by the geometry of the feature $\mathbf{f}_i$. The expected loss across activations $\mathbf{x}$ is then:

$$\mathbf{E}_{\mathbf{x}} [\mathcal{L}(\mathbf{x})] = \mathbf{E}_{\mathbf{x}} \left[\sum_{i \in \mathcal{A}(\mathbf{x})} \mathcal{L}(\mathbf{S}_i \mathbf{f}_i) \right] = \sum_i p_i \mathcal{L}(\mathbf{S}_i \mathbf{f}_i) = \boxed{\sum_i p_i L_i(n_i)}$$

We therefore reduce the SAE optimization problem to the problem of choosing how many latents n_i to allocate to each feature i in the data. To understand SAE scaling behavior, we solve for the allocation of latents to each feature n_i that minimizes $\sum_i p_i L_i(n_i)$ given a distribution over feature frequencies p_i , the per-feature loss curves L_i , and the constraint on the total number of latents $\sum_i n_i = N$.

10.2.3 Warm-up on discrete features

To get comfortable with this formulation, we first apply it to the simplest case where all features are *discrete*. We imagine that for all features i , $d_i = 1$ and $\mathbf{f}_i = 1$ on the fraction p_i of samples where feature i is present and is 0 otherwise. For each feature i , an SAE incurs loss 1 if $n_i = 0$ (reconstruction loss = 1 and sparsity loss = 0). If $n_i = 1$, it can perfectly reconstruct the feature, incurring sparsity loss λ with L0 or L1 sparsity loss. Therefore $L(n) = 1$ if $n_i < 1$ else λ .

With total loss $\sum_i p_i L(n_i)$ and N latents in our SAE we see that the optimal solution is to allocate one latent to the most commonly occurring N features. In this setup, then, **the effect of scaling SAEs is to learn an increasing number of discrete features in the data, in decreasing order of their frequency**. We will often be interested in the number of features “discovered” by the SAE $D(N) = |\{i : n_i > 0\}|$. In this setting, $D(N) = N$. To recover the power law scaling that SAEs exhibit empirically, we only need to assume that $p_i \propto i^{-(1+\alpha)}$. With this assumption, the improvement in total loss from adding a marginal latent i follows $p_i(1 - \lambda)$, and integrating from $i = 1 \dots N$ we get that the total loss drops off as a power law $\mathcal{L}(N) \propto N^{-\alpha}$. We note that

A model of SAE scaling on feature manifolds

If SAEs allocate latents to solve the following optimization problem...

$$\min_{n_1, n_2, \dots} \mathcal{L} = \sum_i p_i L(n_i) \quad \text{s.t.} \quad \sum_i n_i = N,$$

where $p_i \propto i^{-(1+\alpha)}$, $L(n) \propto n^{-\beta}$.

then, as in the neural scaling model from Brill (2024), there are two regimes:

Good regime: $\alpha < \beta$ 😊

$$\mathcal{L} \propto N^{-\alpha}$$

Number of manifolds discovered:

$$D(N) \propto N$$

Pathological regime: $\beta < \alpha$ 😞

$$\mathcal{L} \propto N^{-\beta}$$

Number of manifolds discovered:

$$D(N) \propto N^{\frac{1+\beta}{1+\alpha}}$$

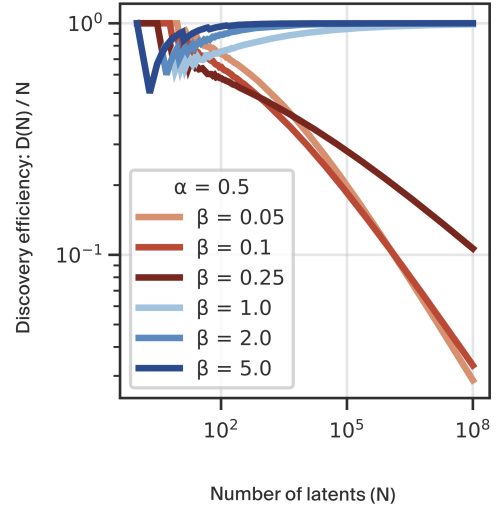


Figure 10.2: Left: Application of Brill’s (2024) [25] capacity-allocation model to SAE scaling. Right: numerical simulation of SAE scaling when the most frequently occurring feature is a manifold with loss scaling as $L(n) \propto n^{-\beta}$ and all other features are discrete. We see that if $\beta \ll \alpha$, then a simulated SAE with 100 million latents discovers only 3 million features.

this model of SAE scaling mirrors the “quanta” model of neural scaling from [28], where here the “quanta” are features. This picture also agrees with the finding from [15] that SAEs learn features for concepts in data approximately in order of the frequencies (roughly Zipfian) at which those concepts occur.

10.2.4 Intuition behind pathological manifold scaling

Recently, several works have commented on the existence of *feature manifolds* [12,23] (multi-dimensional features) [22] in neural networks. In our formalism above, these are features i with $d_i > 1$ and where the range of $\mathbf{f}_i$ is a manifold embedded in $\mathbb{R}^{d_i}$. With feature manifolds, instead of having a discrete $L_i(n_i)$ curve like above, L_i might drop off slowly as n_i grows. To show that this is possible, in Figure 10.1, we show the scaling curve for ReLU SAEs ($\sigma = \text{ReLU}$) trained with L1 penalty to reconstruct points on a circle $\mathbf{x} \in S^1 \subseteq \mathbb{R}^2$. We observe that these SAEs can gradually reduce their total loss by more finely “tiling” the manifold with latents that activate more sparsely, and that this manifold can accommodate dozens of latents before loss plateaus. While such solutions ruthlessly minimize the SAE loss, it is not obvious that they would be better from an interpretability standpoint. Our fundamental concern is that if $L_i(n_i)$ curves decrease gradually, it could be optimal for an SAE to tile common feature manifolds instead of discovering rarer features in data.

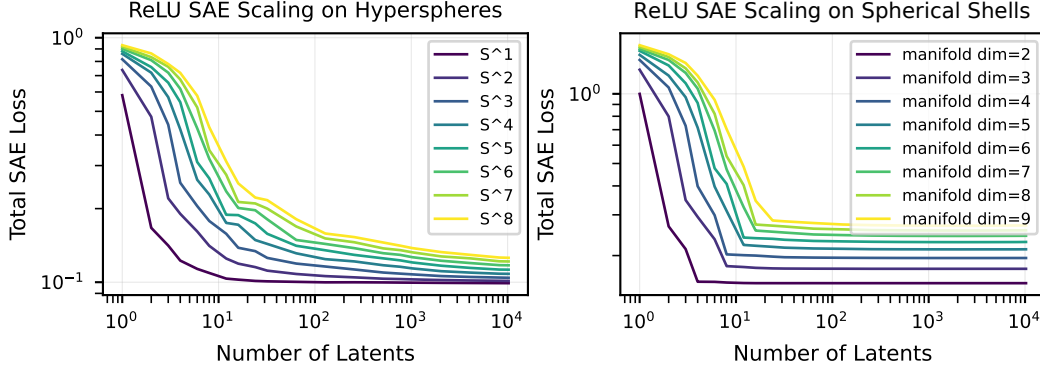


Figure 10.3: ReLU SAE scaling on individual toy feature manifolds, showing how $L(n)$ curves depend on feature geometry. We train with L1 sparsity $\lambda = 0.1$. **Left:** SAE scaling on data drawn from a single unit hypersphere. **Right:** SAE scaling on data sampled in $\{\mathbf{x} : 0.5 < |\mathbf{x}| < 2\}$.

10.2.5 Solution for power-law $L(n)$ following Brill (2024)

To model SAE scaling in the presence of feature manifolds, we assume that feature frequencies decay as a power law $p_i \propto i^{-(1+\alpha)}$. For mathematical simplicity, we also assume that all features have the same power law per-feature loss curve $L_i(n_i) = n_i^{-\beta}$. In this setting, our model of SAE scaling directly corresponds to the model of neural scaling from Brill (2024) [25], where neural networks allocate units of capacity n_i towards approximating functions on distinct power-law distributed data manifolds, with per-manifold loss scaling as $n_i^{-c/D}$. We show similar derivations to Brill [25] in Section 10.5.2 and summarize the core results in Figure 10.2 (left).

In our notation, there is a key threshold: $\alpha < \beta$. When $\alpha < \beta$, expected loss scales as $\mathcal{L}(N) \propto N^{-\alpha}$ like in the discrete feature case, and $D(N) \propto N$, so the “efficiency” at which SAEs discover features $D(N)/N$ tends towards a constant. However, when $\beta < \alpha$, then $\mathcal{L}(N) \propto N^{-\beta}$ and $D(N) \propto N^{\frac{1+\beta}{1+\alpha}}$, and so $D(N)/N$ tends towards 0. This illustrates the core dynamic of concern: when SAEs can continue to reduce loss by “tiling” common feature manifolds, then it can be optimal for them to do this at the expense of discovering other, rarer features.

10.2.6 Numerical simulation

While our mathematical model above assumed that all features have the same $L_i(n_i)$ curve, we can run numerical simulations with arbitrary heterogeneous $L_i(n_i)$ curves. In Figure 10.2 (right), we conduct simulations where the first feature scales as $n^{-\beta}$ but all other features are discrete and scale as step functions $\mathbf{1}_{n=0}$, and with $p_i \propto i^{-(1+0.5)}$. We find that, with only a single feature with power-law $L(n_i)$ scaling, when $\beta < \alpha$ this feature begins to absorb the vast majority of latents in the SAE once N is large.

10.3 SAE scaling on synthetic features and on real neural networks

In our analysis, whether SAEs will scale pathologically depends on the shape of their per-feature loss curves $L_i(n_i)$. We showed one such $L(n)$ curve in Figure 10.1, but we further explore this by training SAEs on a variety of synthetic manifolds in Figure 10.3 and in Appendix Figure 10.6. Overall, we find that the shape of the $L(n)$ curve depends on the manifold geometry, with some manifolds accommodating many thousands of latents without saturating loss while on others the loss curve plateaus very quickly (effectively $\beta \rightarrow \infty$). In Section 10.5.1, we provide further discussion on what α and β may be, and whether SAEs might be scaling pathologically, in practice.

When a large number of SAE latents are allocated to a relatively low-dimensional feature manifold, we’d expect the cosine similarities between the decoder latents allocated to that manifold to be close to 1. In Section 10.5.5, we show the distribution over decoder latent nearest neighbor cosine similarities for SAEs trained on LLM and vision model activations, and find some differences between them.

10.4 Discussion

In this short paper, we have described a model of SAE scaling which reduces the SAE optimization problem to the problem of optimally allocating different numbers of SAE latents to different features in data. As in the model of scaling from Brill [25] SAE scaling laws either result from an underlying power law distribution over features $p_i \propto i^{-(1+\alpha)}$, or from the improvements in loss from tiling common feature manifolds following a power law $L_i(n_i) \propto n_i^{-\beta}$. When $\beta < \alpha$, SAE latents could massively accumulate on commonly occurring feature manifolds.

Unfortunately, we do not resolve the question of whether SAEs are in this pathological scaling regime in practice. Our uncertainty is due to our knowing neither the distribution over true feature frequencies (determining α) nor the geometry of real-world neural network feature manifolds (which determines β). We give some additional commentary in Section 10.5.1. Arguably our most important observation is that when we train SAEs on toy feature manifolds with variation in the radial direction, our SAEs do not tile the manifold and instead exhibit $L(n)$ curves which plateau early. If this geometry is realistic [22,29], then manifolds may not pose an issue to SAE scaling in practice. Overall though, we view this work as primarily about framing, rather than completely answering, this interesting problem.

Acknowledgements

We thank Owen Lewis, Michael Pearce, Jack Merullo, Dan Balsam, Michael Byun, Elana Simon, and Wes Gurnee for helpful conversations and feedback. EJM was supported by the NSF via the Graduate Research Fellowship Program (Grant No.2141064). This work was done at [Goodfire AI](#) in San Francisco.

References

- [1] A. Ng et al. “Sparse autoencoder.” *CS294A Lecture notes*, 72(2011), 2011, pp. 1–19.
- [2] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey. “Sparse autoencoders find highly interpretable features in language models.” *arXiv preprint arXiv:2309.08600*, 2023.
- [3] T. Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning.” *Transformer Circuits Thread*, 2023. URL: <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [4] L. Gao, T. D. la Tour, H. Tillman, G. Goh, R. Troll, A. Radford, I. Sutskever, J. Leike, and J. Wu. “Scaling and evaluating sparse autoencoders.” *arXiv preprint arXiv:2406.04093*, 2024.
- [5] S. Rajamanoharan, T. Lieberum, N. Sonnerat, A. Conmy, V. Varma, J. Kramár, and N. Nanda. “Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders.” *arXiv preprint arXiv:2407.14435*, 2024.
- [6] B. Bussmann, P. Leask, and N. Nanda. “Batchtopk sparse autoencoders.” *arXiv preprint arXiv:2412.06410*, 2024.
- [7] T. Fel, E. S. Lubana, J. S. Prince, M. Kowal, V. Boutin, I. Papadimitriou, B. Wang, M. Wattenberg, D. Ba, and T. Konkle. “Archetypal sae: Adaptive and stable dictionary learning for concept extraction in large vision models.” *arXiv preprint arXiv:2502.12892*, 2025.
- [8] M. Muchane, S. Richardson, K. Park, and V. Veitch. “Incorporating Hierarchical Semantics in Sparse Autoencoder Architectures.” *arXiv preprint arXiv:2506.01197*, 2025.
- [9] V. Costa, T. Fel, E. S. Lubana, B. Tolooshams, and D. Ba. “From Flat to Hierarchical: Extracting Sparse Representations with Matching Pursuit.” *arXiv preprint arXiv:2506.03093*, 2025.
- [10] J. Dunefsky, P. Chlenski, and N. Nanda. “Transcoders find interpretable llm feature circuits.” *Advances in Neural Information Processing Systems*, 37, 2024, pp. 24375–24410.
- [11] J. Lindsey, A. Templeton, J. Marcus, T. Conerly, J. Batson, and C. Olah. *Sparse Crosscoders for Cross-Layer Features and Model Diffing*. Research update. Oct. 2024. URL: <https://transformer-circuits.pub/2024/crosscoders/index.html>.
- [12] L. Gorton. “Group crosscoders for mechanistic analysis of symmetry.” 2024. arXiv: [2410.24184](https://arxiv.org/abs/2410.24184).
- [13] J. Minder, C. Dumas, C. Juang, B. Chughtai, and N. Nanda. “Robustly identifying concepts introduced during chat fine-tuning using crosscoders.” *arXiv preprint arXiv:2504.02922*, 2025.
- [14] E. Ameisen et al. “Circuit Tracing: Revealing Computational Graphs in Language Models.” *Transformer Circuits Thread*, 2025. URL: <https://transformer-circuits.pub/2025/attribution-graphs/methods.html>.
- [15] A. Templeton et al. “Scaling Monosemanticity: Extracting Interpretable Features from Claude 3 Sonnet.” *Transformer Circuits Thread*, 2024. URL: <https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html>.
- [16] T. Lieberum, S. Rajamanoharan, A. Conmy, L. Smith, N. Sonnerat, V. Varma, J. Kramár, A. Dragan, R. Shah, and N. Nanda. “Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2.” *arXiv preprint arXiv:2408.05147*, 2024.

- [17] J. Lindsey et al. “On the Biology of a Large Language Model.” *Transformer Circuits Thread*, 2025. URL: <https://transformer-circuits.pub/2025/attribution-graphs/biology.html>.
- [18] J. Lindsey, T. Conerly, A. Templeton, J. Marcus, and T. Henighan. *Scaling Laws for Dictionary Learning*. Circuits Updates — April 2024. Apr. 2024. URL: <https://transformer-circuits.pub/2024/april-update/index.html#scaling-laws>.
- [19] C. Olah and A. Jermyn. *The Dark Matter of Neural Networks?* Transformer Circuits Thread. Part of Circuits Updates - July 2024, Anthropic Interpretability Team. July 2024. URL: <https://transformer-circuits.pub/2024/july-update/index.html#dark-matter>.
- [20] R. Csordás, C. Potts, C. D. Manning, and A. Geiger. “Recurrent neural networks learn to store and generate sequences using non-linear representations.” *arXiv preprint arXiv:2408.10920*, 2024.
- [21] S. S. R. Hindupur, E. S. Lubana, T. Fel, and D. Ba. “Projecting assumptions: The duality between sparse autoencoders and concept geometry.” *arXiv preprint arXiv:2503.01822*, 2025.
- [22] J. Engels, E. J. Michaud, I. Liao, W. Gurnee, and M. Tegmark. “Not all language model features are one-dimensionally linear.” *arXiv preprint arXiv:2405.14860*, 2024.
- [23] A. Modell, P. Rubin-Delanchy, and N. Whiteley. “The Origins of Representation Manifolds in Large Language Models.” *arXiv preprint arXiv:2505.18235*, 2025.
- [24] J. Engels, L. Riggs, and M. Tegmark. “Decomposing the dark matter of sparse autoencoders.” *arXiv preprint arXiv:2410.14670*, 2024.
- [25] A. Brill. “Neural scaling laws rooted in the data distribution.” *arXiv preprint arXiv:2412.07942*, 2024.
- [26] N. Elhage et al. “Toy Models of Superposition.” *Transformer Circuits Thread*, 2022. URL: https://transformer-circuits.pub/2022/toy%5C_model/index.html.
- [27] D. Chanin, J. Wilken-Smith, T. Dulka, H. Bhatnagar, S. Golechha, and J. Bloom. “A is for absorption: Studying feature splitting and absorption in sparse autoencoders.” *arXiv preprint arXiv:2409.14507*, 2024.
- [28] E. Michaud, Z. Liu, U. Girit, and M. Tegmark. “The quantization model of neural scaling.” *Advances in Neural Information Processing Systems*, 36, 2023, pp. 28699–28722.
- [29] C. Olah. *What is a Linear Representation? What is a Multidimensional Feature?* Transformer Circuits Thread. Part of Circuits Updates - July 2024, Anthropic Interpretability Team. July 2024. URL: <https://transformer-circuits.pub/2024/july-update/index.html#linear-representations>.
- [30] D. Till. *Do sparse autoencoders find “true features”?* <https://www.lesswrong.com/posts/QoR8noAB3Mp2KBA4B/do-sparse-autoencoders-find-true-features>. Accessed: 2025-08-11. Feb. 2024.
- [31] E. Anders, C. Neo, J. Hoelscher-Obermaier, and J. N. Howard. *Sparse autoencoders find composed features in small toy models*. <https://www.lesswrong.com/posts/a5wwqza2cY3W7L9cj/sparse-autoencoders-find-composed-features-in-small-toy>. Accessed: 2025-08-11. Mar. 2024.

- [32] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling laws for neural language models.” *arXiv preprint arXiv:2001.08361*, 2020.
- [33] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. “Training compute-optimal large language models.” *arXiv preprint arXiv:2203.15556*, 2022.
- [34] T. Conerly, A. Templeton, T. Bricken, J. Marcus, and T. Henighan. *Update on how we train SAEs*. Transformer Circuits Thread. Part of Circuits Updates - April 2024, Anthropic Interpretability Team. Apr. 2024. URL: <https://transformer-circuits.pub/2024/april-update/index.html#training-saes>.
- [35] T. Conerly, H. Cunningham, A. Templeton, J. Lindsey, B. Hosmer, and A. Jermyn. *Dictionary Learning Optimization Techniques*. Transformer Circuits Thread. Part of Circuits Updates - January 2025, Anthropic Interpretability Team. Jan. 2025. URL: <https://transformer-circuits.pub/2025/january-update/index.html#DL>.

10.5 Appendix

10.5.1 Additional discussion

Are real SAEs in the pathological regime?

It is worth attempting to say more about whether SAEs are in the pathological scaling regime in practice. As we stated in Section 10.2.5, this depends on the rate at which the feature frequencies $p_i \propto i^{-(1+\alpha)}$ decay vs. the rate at which the per-feature SAE loss decays $L(n_i) \propto i^{-\beta}$. In Section 10.5.2, following Brill (2024) [25], we show that when $\alpha < \beta$, the efficiency at which SAEs discover features $D(N)/N$ approaches a reasonable constant, but when $\beta < \alpha$, $D(N)/N$ approaches 0 as $N \rightarrow \infty$. Whether feature manifolds could cause pathological SAE scaling in the real world depends then on the real α (assuming it’s even a power law) and β (assuming $L_i(n_i)$ is also a power law $\propto n_i^{-\beta}$).

What is α ?: We first speculate on what α may be. One way of trying to measure this is to look at how the latent activation frequencies decay when sorted by frequency. For a few Gemma Scope SAEs [16], we show these curves in Figure 10.4, and measure slopes between -0.57 and -0.74 . If there was a one-to-one relationship between SAE latents and features in the data, then this would imply an $\alpha \approx 0.5$ to $\alpha \approx 0.7$. However, feature absorption [27], the learning of compositional features [30,31], and latents tiling a feature manifold could distort the relationship between the true feature frequencies and the latent activation frequencies.

Another highly speculative way of trying to estimate α could be to look at the exponents of neural scaling laws for models like those the SAE is being trained on. The idea here is that if the “features” are the computational units of neural networks that ref [28] called the “quanta”, then the underlying neural scaling law slope would reflect the distribution over feature occurrences. Neural scaling exponents for language models (w.r.t. network parameters) (α_N in the scaling law $N^{-\alpha_N}$ have been measured to have an α_N of 0.07 [32] and 0.34 [33], potentially implying a distribution over quanta/features $p_i \propto i^{-(1+\alpha)}$ with exponent α in that same range.

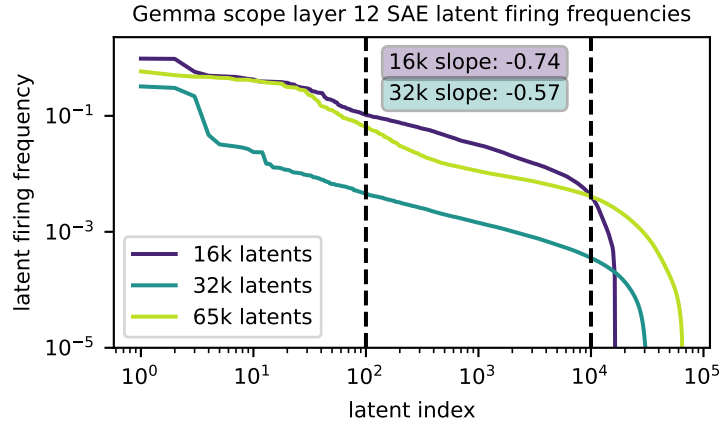


Figure 10.4: Frequencies at which latents fire in gemma scope SAEs, sorted by frequency. We measure the power law decay exponent between latent 10^2 and 10^4 to be -0.74 for an SAE with 16k latents and -0.57 for an SAE with 32k latents.

What is β ? In Figure 10.3 and Figure 10.6, we show $L(n)$ scaling curves for SAEs reconstructing single synthetic feature manifolds. We find that these curves depend on the feature geometry. In particular, we see that when reconstructing hollow hyperspheres, we can observe gradual $L(n)$ scaling. The higher-dimensional hyperspheres in particular can accommodate at least 10^4 latents without loss plateauing. In Figure 10.5, we plot the slope that we measure for these curves between 10^2 and 10^4 latents, and measure a β of roughly 0.05 for hyperspheres with dimension 6-8.

However, in the more realistic setting where there is variation in the radial direction—the intensity that features fire [29]—we see that manifolds tend to saturate very quickly. It appears that the saturation happens when $n_i \approx 2d_i$, likely corresponding to solutions where the SAE latents form a basis for the subspace where the feature is embedded (or rather use two latents for each basis direction, one in the “positive” direction and one in the “negative” direction since latents can only fire positively).

Therefore, the slope of the $L_i(n_i)$ curve, and whether SAEs can use a large number of latents to reduce loss on feature manifolds, depends on the geometry of the manifold. Our experiments so far indicate that in the more realistic setting where there is variation in the radial direction (which was seen in practice in ref [22]), that SAEs do not discover solutions which take advantage of a large number of latents, and instead learn a basis solution. **This is probably the strongest argument against the possibility of feature manifolds causing pathological SAE scaling.**

Lastly, we note that we are unsure how “ripples” [29] in feature manifolds could affect SAE scaling on them. If a feature manifold is intrinsically low dimensional, but ripples through a large number of other dimensions, we could imagine SAE solutions potentially looking different.

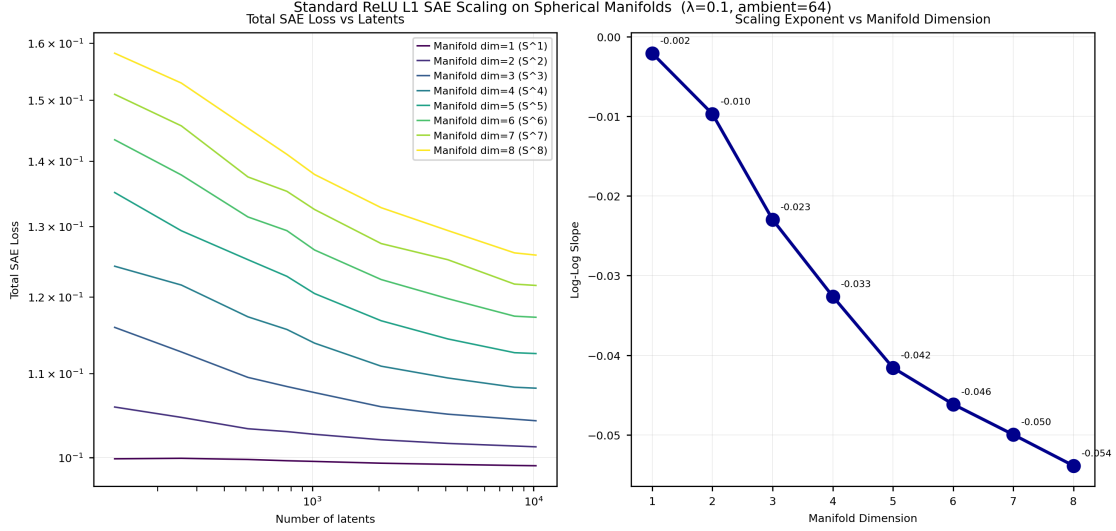


Figure 10.5: Measured slopes of L_i curves for ReLU L1 SAEs trained on hyperspheres.

10.5.2 Derivations

Loss decomposition into per-feature terms

We work under the assumptions in §10.2.2: (i) *feature-specific latents*—each latent j fires only when a unique feature $i(j)$ is active; (ii) *decoder respect for subspaces*—decoder columns for latents assigned to feature i lie in $\text{span}(\mathbf{S}_i)$; and (iii) *orthogonal feature subspaces*— $\text{span}(\mathbf{S}_i) \perp \text{span}(\mathbf{S}_k)$ for $i \neq k$.

Additivity of sparsity. For a sample $\mathbf{x} = \sum_{i \in \mathcal{A}(\mathbf{x})} \mathbf{S}_i \mathbf{f}_i$, feature-specificity implies $\hat{\mathbf{f}}_j(\mathbf{x}) = 0$ unless $i(j) \in \mathcal{A}(\mathbf{x})$. For separable sparsity penalties (L0/L1), $S(\hat{\mathbf{f}}) = \sum_j s(\hat{f}_j)$, so

$$S(\hat{\mathbf{f}}(\mathbf{x})) = \sum_{i \in \mathcal{A}(\mathbf{x})} \sum_{j: i(j)=i} s(\hat{f}_j(\mathbf{x})).$$

Thus the sparsity cost splits across active features.

Orthogonal reconstruction. Write the model reconstruction as $\hat{\mathbf{x}} = \sum_i \hat{\mathbf{x}}_i$, with $\hat{\mathbf{x}}_i := \sum_{j: i(j)=i} \mathbf{w}_j \hat{f}_j \in \text{span}(\mathbf{S}_i)$ by (ii). Then, using (iii) and the Pythagorean theorem,

$$\|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 = \left\| \sum_i (\mathbf{S}_i \mathbf{f}_i - \hat{\mathbf{x}}_i) \right\|_2^2 = \sum_i \|\mathbf{S}_i \mathbf{f}_i - \hat{\mathbf{x}}_i\|_2^2.$$

Hence the per-sample objective $\mathcal{L}(\mathbf{x}) = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 + \lambda S(\hat{\mathbf{f}})$ decomposes as a sum over features active on that sample. Taking expectations and letting n_i be the number of latents allocated to feature i , we obtain

$$\mathbb{E}_{\mathbf{x}}[\mathcal{L}(\mathbf{x})] = \sum_i p_i L_i(n_i), \quad \sum_i n_i = N \quad (10.1)$$

where $p_i \equiv \Pr[\mathbf{f}_i \neq 0]$ and $L_i(n_i)$ is the optimal (feature- i) expected reconstruction-plus-sparsity loss achieved with n_i latents restricted to $\text{span}(\mathbf{S}_i)$. We note that the derivations below closely follow those in Brill (2024) [25], adapted to our SAE setting with appropriate changes in notation and interpretation. We show them here for convenience.

Scaling setup and notation

We study the optimal latent allocation n_i minimizing (10.1) under two empirical power-law regularities:

$$p_i \propto i^{-(1+\alpha)} \quad (\text{features sorted by frequency}), \quad L_i(n) \equiv L(n) \propto n^{-\beta},$$

with $\alpha, \beta > 0$.¹ Define the discovery count

$$D(N) \equiv |\{i : n_i > 0\}|$$

and the total expected loss $\mathcal{L}(N) \equiv \sum_i p_i L(n_i)$ at total width N . A standard Lagrange multiplier treatment (continuous relaxation) yields

$$n_i \propto p_i^{\frac{1}{1+\beta}} \propto i^{-\gamma}, \quad \gamma \equiv \frac{1+\alpha}{1+\beta}. \quad (10.2)$$

In practice there is a cutoff index i_c (“last discovered feature”) with $n_{i_c} \approx 1$ and $n_i \lesssim 1$ for $i > i_c$. Then $D(N) \asymp i_c$ and

$$N = \sum_{i \leq i_c} n_i \propto \sum_{i \leq i_c} i^{-\gamma}.$$

Two regimes follow depending on whether the allocation tail-sum diverges or converges.

Case $\beta < \alpha$ (simple, latent accumulation on frequent features)

Here $\gamma > 1$, so $\sum_{i \geq 1} i^{-\gamma}$ converges to a constant $Z(\gamma)$. From (10.2), $N \propto \sum_{i \leq i_c} i^{-\gamma} \rightarrow Z(\gamma)$ implies the proportionality constant in (10.2) scales as $\kappa \propto N$. The discovery cutoff is set by $n_{i_c} \approx 1$:

$$1 \approx \kappa i_c^{-\gamma} \implies i_c \propto \kappa^{1/\gamma} \propto N^{\frac{1}{\gamma}} = N^{\frac{1+\beta}{1+\alpha}}.$$

Thus

$$\boxed{D(N) \propto N^{\frac{1+\beta}{1+\alpha}} \quad (\text{sublinear discovery}).} \quad (10.3)$$

For the loss, the discovered part scales as

$$\sum_{i \leq i_c} p_i n_i^{-\beta} \propto \kappa^{-\beta} \sum_{i \leq i_c} i^{-\gamma} \propto \kappa^{-\beta} \propto N^{-\beta},$$

while the undiscovered tail $\sum_{i > i_c} p_i \propto i_c^{-\alpha} \propto N^{-\alpha(1+\beta)/(1+\alpha)}$ decays faster since $\alpha > \beta$. Therefore

$$\boxed{\mathcal{L}(N) \propto N^{-\beta}.} \quad (10.4)$$

Intuitively, the SAE keeps shaving loss on common feature manifolds; discovery lags.

¹Constants are inessential for power-law exponents and are dropped.

Case $\alpha < \beta$ (benign, feature discovery keeps up)

Now $\gamma < 1$, so $\sum_{i \leq i_c} i^{-\gamma} \propto i_c^{1-\gamma}$. Using $N \propto \kappa i_c^{1-\gamma}$ and the threshold $1 \approx \kappa i_c^{-\gamma}$, we eliminate κ to find

$$N \propto i_c \quad \implies \quad \boxed{D(N) \propto N.}$$

For the loss over discovered features,

$$\sum_{i \leq i_c} p_i n_i^{-\beta} \propto \kappa^{-\beta} \sum_{i \leq i_c} i^{-\gamma} \propto i_c^{-\beta\gamma} i_c^{1-\gamma} = i_c^{1-(1+\alpha)} = i_c^{-\alpha} \propto N^{-\alpha}.$$

The undiscovered tail obeys $\sum_{i > i_c} p_i \propto i_c^{-\alpha} \propto N^{-\alpha}$, so both pieces match and

$$\boxed{\mathcal{L}(N) \propto N^{-\alpha}.} \tag{10.5}$$

Here, extra width primarily buys new features rather than over-tiling old manifolds; loss scaling mirrors the frequency tail.

10.5.3 Additional Experimental Details

For Figure 10.3 and Figure 10.6, we trained SAEs on synthetic feature manifolds. For these experiments, we trained for 12000 steps, with a batch size of 2048, and a learning rate of 10^{-3} with the Adam optimizer. For the L1 penalty calculation, we use the trick of multiplying the decoder vector L2 norms by the latent activation [34].

10.5.4 SAE scaling curves on synthetic manifolds

In Figure 10.6, we show JumpReLU SAE scaling on individual feature manifolds like we did for ReLU SAEs in Figure 10.3.

10.5.5 SAE feature geometry on LLMs and vision models

JumpReLU Gemma Scope SAEs

If a large number of latents “tile” a single low-dimensional feature manifold, then we would expect the decoder directions for those SAE latents to have neighbors with high cosine similarity. One can see this effect directly in Figure 10.1, where we see that the SAE decoder latents begin to be arranged quite tightly together along the manifold. In this section, we study whether SAEs on real neural network activations have large numbers of latents with very high cosine similarity to their nearest neighbor.

We first study this in the Gemma Scope SAEs [16]. In Figure 10.7, we plot the distribution over cosine similarities between decoder latent vectors and their nearest neighbor for Gemma Scope SAEs on layer 12 (residual stream) of gemma-2-2b. While the distribution is skewed substantially higher than one would expect if all latent decoder vectors were random (and thus approximately orthogonal), we do not overall see a very large fraction of latents with extremely high cosine similarity to their nearest neighbor.

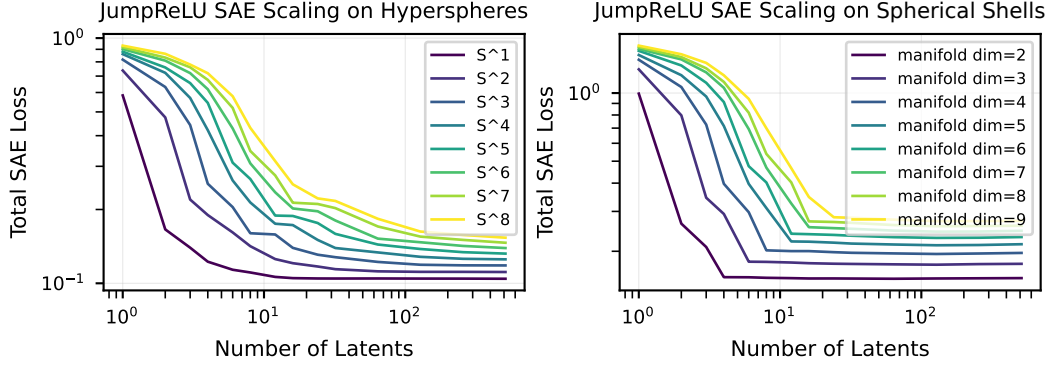


Figure 10.6: JumpReLU SAE scaling on individual toy feature manifolds, showing how $L(n)$ curves depend on feature geometry. We train with the tanh loss from [35] with $c = 0.1$ and $\lambda_s = 1.0$. **Left:** JumpReLU SAE scaling on unit hyperspheres of varying dimension. **Right:** JumpReLU SAE scaling on points sampled in $\{\mathbf{x} : 0.5 < |\mathbf{x}| < 2\}$. We see that, like with ReLU SAEs, that when there is variation in the radial direction between samples that our SAEs do not learn solutions which can continue to accommodate latents, and instead plateau after allocating roughly $2d_i$ latents to the feature manifold.

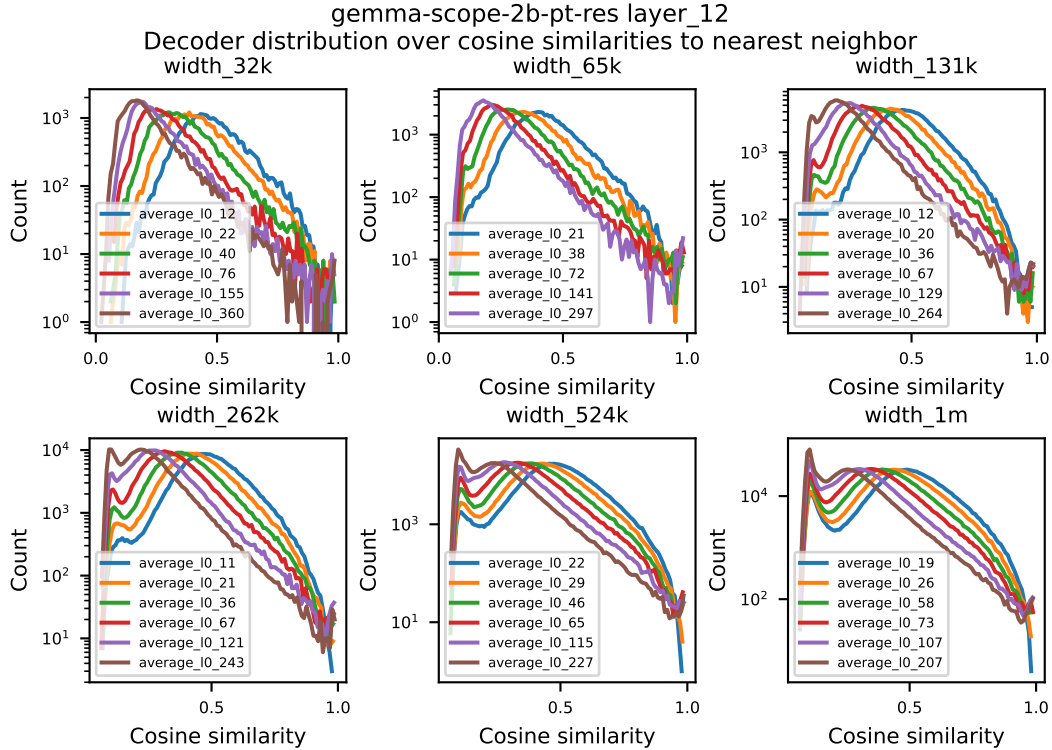


Figure 10.7: Distribution over cosine similarities between decoder vectors and their nearest neighbor.

Intriguingly though, for some SAEs we do see a small uptick on the right side of this distribution, where between 10-100 latents have cosine similarity > 0.97 with their nearest neighbor. When we

investigated these latents in one SAE (width_262k, average_l0_121), we found that for each of these latents, their nearest neighbor was dead (across a dataset of over 250 million tokens). These latents are not then being used to very sparsely reconstruct points on a manifold, and instead seem to be an artifact of the training process. However, the alive latents in this set are not typical SAE latents. A large number of these latents fire on tokens representing single numerical digits and single alphabet characters. We do not have an explanation of this phenomenon, but wonder whether there may be some underlying manifold representation which the SAE at one point in training tried to “tile”, but then when the latents got too close, one of them was killed to reduce the L0 loss.

ReLU L1 SAEs on Inception-v1

We also study the geometry of latent decoder directions on SAEs trained on Inception-v1 activations. We train SAEs on activations from mixed3b using an L1 coefficient, λ , of 1, a learning rate of 10^{-4} , and an expansion factor of 16 (a total of 7680 latents).

In Figure 10.8, we find that on Inception-v1, a meaningful fraction of SAE latents have very high cosine similarity with their nearest neighbor. We note however that this could be due to latents being duplicated, which is not strongly disincentivized by the L1 loss, as pointed out in [13].

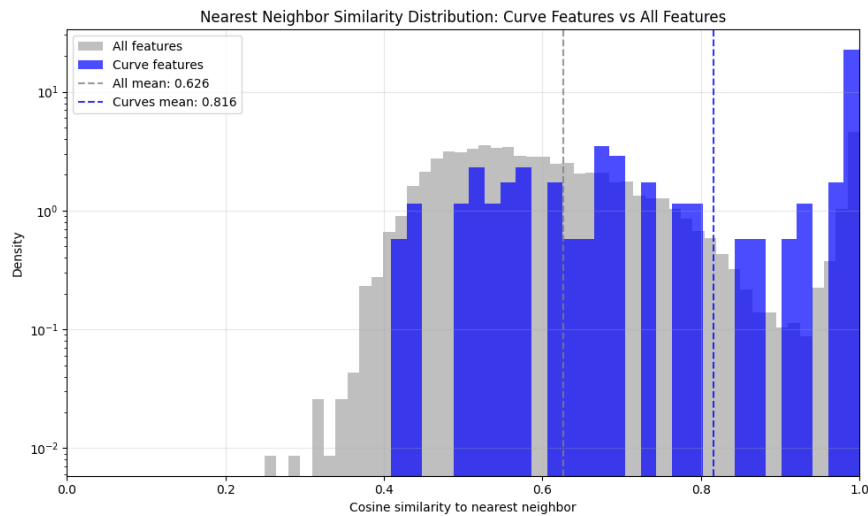


Figure 10.8: Distribution over pairwise cosine similarities for Inception V1.