Grokking is fast in transformers

April 27, 2026

Abstract

We describe a short series of experiments that highlight the importance of minibatch SGD for accelerating grokking in transformers trained on modular arithmetic tasks. This is important as grokking occurs when the model learns a low-dimensional topology for representing relations between tokens (or vectors) – here the cyclial order of the integers mod(p) – which is a powerful tool for generalization.

1 Introduction

Grokking is the phenomena in machine learning where a model transitions from representing a task (typically modular arithmetic) as a per-instance map that does not generalize (e.g. a lookup table) to a generalizing function (modular addition using Fourier components) [1]. The Fourier solution to modular arithmetic entails the network learning the **order** of the integers, which are supplied as one-hot vocabulary members, from the supplied input-output relations in the supervised data. (At initialization, the one-hots look like randomly-oriented high-D vectors after the embedding matrix.) Learning an ordering is equivalent to learning a geometry (or, more flexibly, a topology) over any domain, which is useful for representation learning beyond integers.

2 MLPs

Many different networks exhibit grokking, but research has focused on notably MLPs and transformers. It was surprising to me that MLPs can also grok, albeit slowly, so I set to see if imbuing the network with explicit knowledge of the topology helps.

Starting with a task $c = (a + b) \bmod (p)$ where p is prime, we need to infer an ordering of the integers a, b, and c; in a MLP, we cannot use the communicative properties of addition, so need to rely on between-sample comparisons. This is possible with 'experiments': hold a constant and sweep b, looking for matches in c:

$$a_1 + b = c$$

$$a_2 + b' = c'$$

If $c = c'$ then we know that b and b' are separated by $\Delta = a_1 - a_2$:

$$a_1 - a_2 + b - b' = 0$$

and therefore must be neighbors. (Δ is a 'generator' over the finite cyclical field p) Once you have an ordering for b, you can infer a concordant mapping for c by simply holding a fixed. A similar construction works for a, going back from (now ordered) c with a fixed b. See cayley.py in the associated repo for an implementation and Figure 1 for a visualization.

This approach can be applied to the encoding layer of an MLP by

1. Calculating the graph Laplacian based on proximity (adjacency per above)

2. Using the graph Laplacian matrix L to regularize the encoding W such that adjacent embeddings are similar[1]:

$$\mathit{topo\_loss} = tr(W^T L W) = \sum_{i=0}^{N} \sum_{j=0}^{N} L_{ij} ||w_i - w_j||_2^2$$

[1] $L = D - W$ where W is the adjacency matrix and D is the node degree; the square on the right, mesuring the L_2 distance between embedding vectors w_i is completed after some algebra.

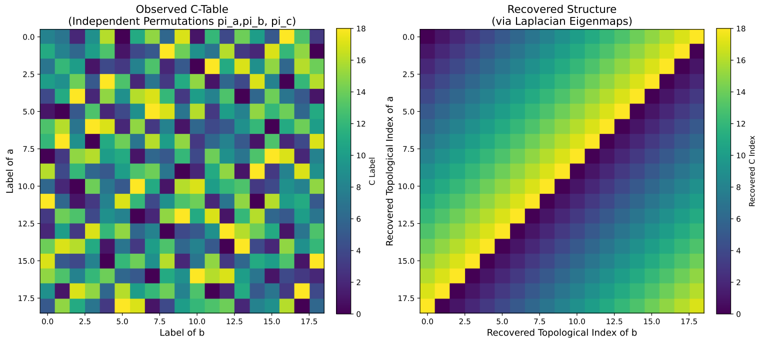

Figure 1: Randomly permuted Cayley table (left) can be unpermuted by inferring neighbors (right)

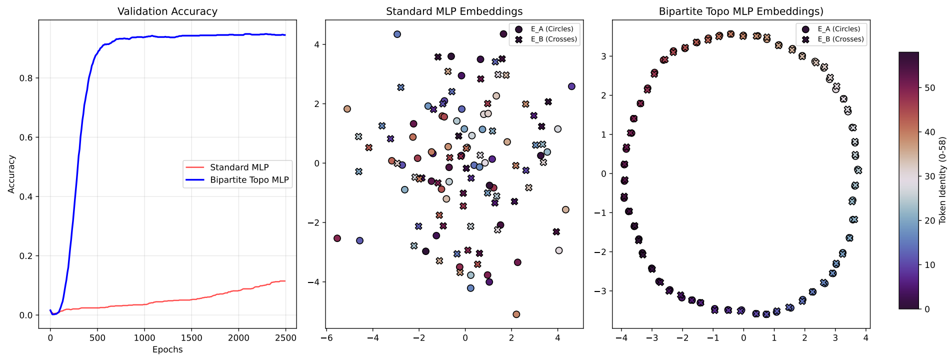

Figure 2: Applying a topological regularization to the input weight matrix accelerates grokking, and causes the resulting embeddings to lie on an approximate circle.

This works well, see topogrok.py and Figure 2

Another avenue for encouraging low-dimensional topology in network weights is to regularize the average gradient outer product (AGOP), which looks at (effectively) the correlation matrix of the gradient of the function ($\frac{\partial output}{\partial input}$) measured at the input points [2]. Minimizing the trace of the AGOP matrix in the input space is effectively a nuclear norm, forcing low-dimensional & ordered representations input encoding (here integers). See agopgrok.py and Figure 3

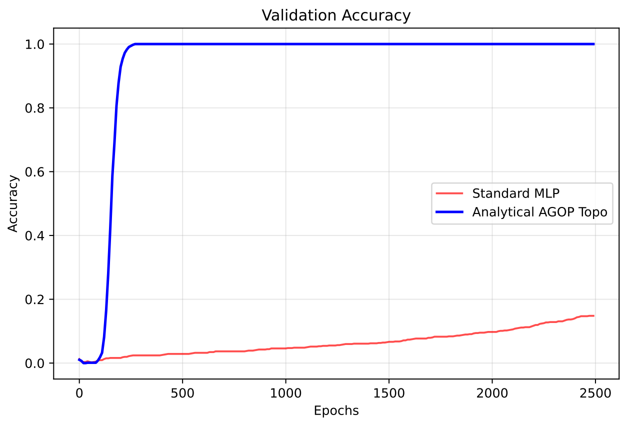

Figure 3: MLP trained with AGOP regularization. Works even better than the graph Laplacian!

3 Transformers

Now, does the AGOP regularization extend to transformers? **It turns out not to matter, as transformers grok very quickly.** This was surprising, as the literature suggests that you need thousands t otens of thousands of epochs to witness grokking. Instead, we found that you typically need ≈ 30 passes through the data. You don't need L_2 regularization (weight decay), or even layer norm; instead, **what's important is minibatch SGD** (as well as training fraction, of course – can't learn an ordering if there are no overlaps in a, b and c).

We tried to replicate the configuration described in [1]: the model is a 1-layer, 1-head transformer, with a ReLU 4x expansion FFN layer, untied embedding and unembedding matrices, and either $p = 59$ or $p = 113$. See fastgrok.py for the implementation, and Figures 4 - 7.

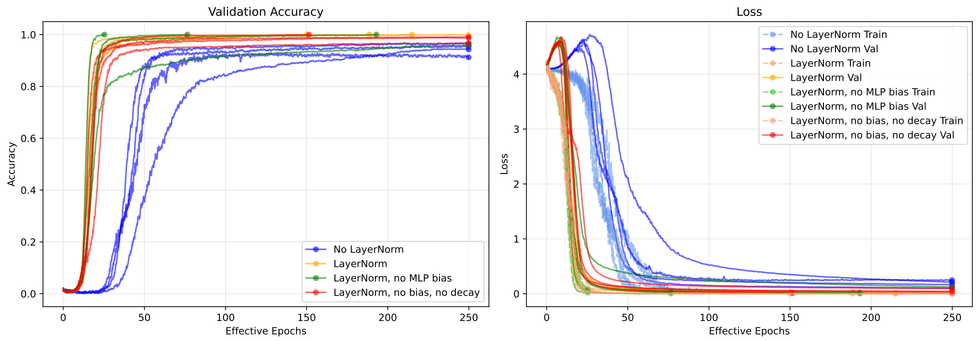

Figure 4: Performance of a transformer trained on modular arithmetic task $(a + b) \bmod (p)$, $p = 59$, 60% training data / 40% validation, randomly drawn & random initialization with 5 replicates and four different network configurations. Effective epochs are the number of times the data (length 2088 in this case) are seen. Batch size is 64. Small circle indicates point on training when validation accuracy reaches 100% Best configuration is with LayerNorm, no bias terms, and weight decay = 0.02. Note the relatively short period of memorization, when the training loss decreases, yet the validation loss increases.

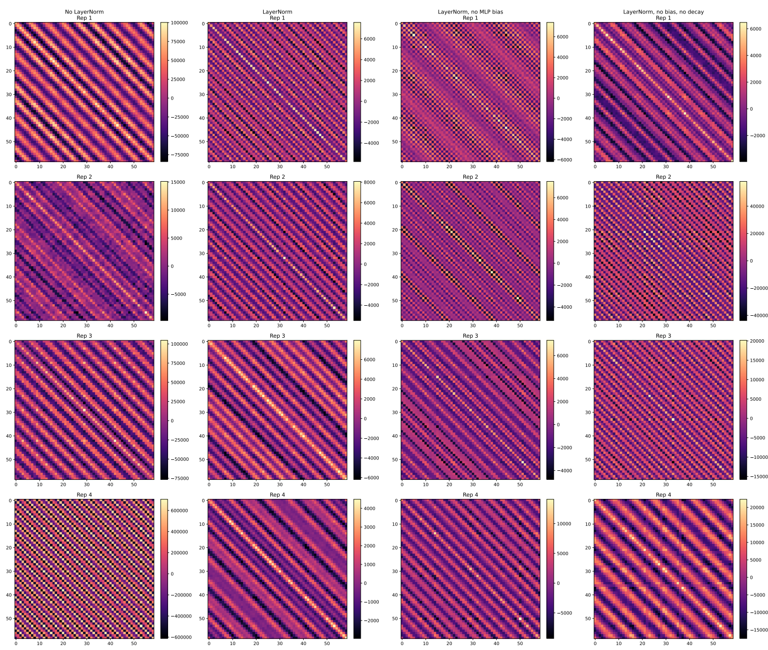

Figure 5: AGOP matrix for the learned models over the 4 different conditions and 5 replicates. The matrices are always block-circulant, as expected for modular arithmetic [2].

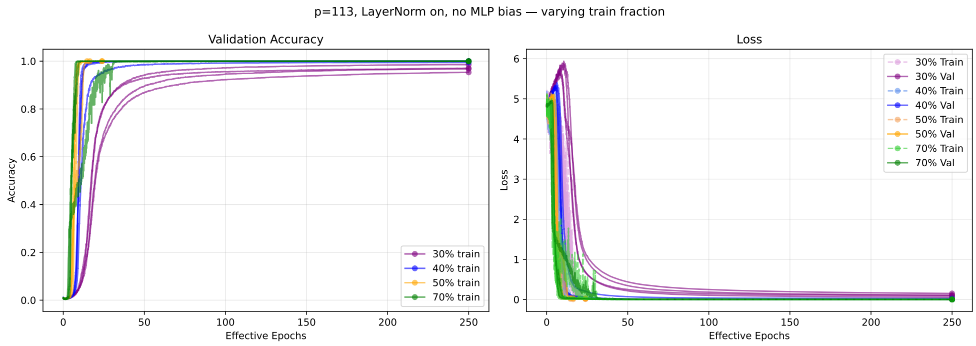

Figure 6: Performance with $p = 113$ and batch size $= 8$, with varying training fractions. Grokking is approx two orders of magnitude faster than reported in literature. Note that in this experiment, both the learning rate and weight decay were decayed with epoch to prevent loss spikes. This prevents the 30% training condition from attaining 100% accuracy through a slingshot mechanism [3].

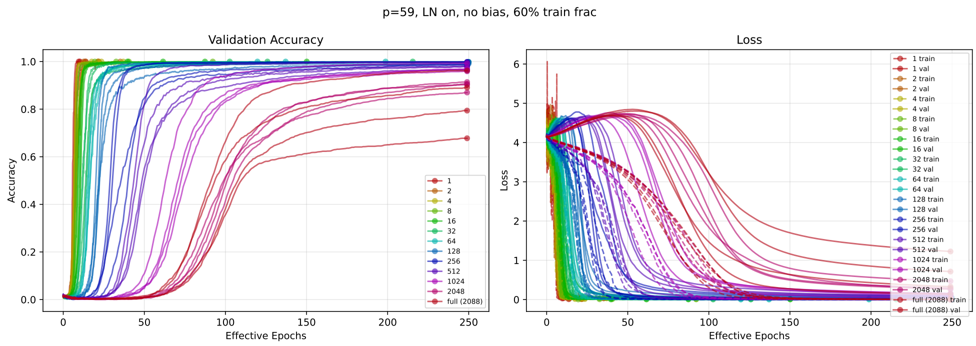

Figure 7: Performance with $p = 59$, train fraction $= 60\%$ over variable batch size. As in the other plots, the circles indicate when 100% validation accuracy is achieved. Clearly, batch size is the critical variable!

4 Recap

What matters is:

- Minibatch SGD. With all else held constant, more gradient steps & less gradient averaging increases grokking speed, approximately linearly.
- Training fraction. A higher fraction increases the number of available measurable neighbors, each which constrain the representation, facilitating topology inference (apologies for not quantifying this).

Timothy Hanson
April 2026

References

[1] N. Nanda, L. Chan, T. Lieberum, J. Smith, and J. Steinhardt. Progress measures for grokking via mechanistic interpretability. http://arxiv.org/abs/2301.05217

[2] N. Mallinar, D. Beaglehole, L. Zhu, A. Radhakrishnan, P. Pandit, and M. Belkin. Emergence in non-neural models: Grokking modular arithmetic via average gradient outer product. http://arxiv.org/abs/2407.20199

[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. http://arxiv.org/abs/2206.04817