Grokking on Small Algorithmic Datasets

A Reproducible Study of Delayed Generalization

Jiaju Wu Kairui Li Kehan Huang

Abstract

Grokking is a delayed generalization phenomenon in which a neural network first memorizes a small algorithmic training set and only much later discovers a rule that transfers to unseen examples. This report studies grokking through a PyTorch codebase for modular arithmetic, polynomial modular operations, symmetric group operations, and K-ary modular summation. The implementation encodes each problem as a short sequence prediction task and supervises only the final answer token. We compare decoder-only Transformers with MLP, LSTM, and GRU baselines, and expose the effects of data fraction, optimizer choice, weight decay, dropout, gradient noise, and task arity. The main conclusion is that grokking is not tied to one architecture. It is better understood as a competition between memorizing a finite table and learning a lower-complexity algebraic rule, with regularization and data sparsity strongly shaping the time scale of the transition.

1 Introduction

Modern neural networks can fit random labels, yet they often generalize well on real tasks. Grokking [1] makes this tension unusually visible. On small algorithmic datasets, a model may reach nearly perfect training accuracy while validation accuracy remains close to random guessing. After many more gradient updates, validation accuracy can rise sharply, sometimes long after ordinary early stopping would have terminated training.

This project investigates grokking in a controlled setting. The base task is modular addition,

$$(x, y) \mapsto (x + y) \bmod p, \qquad x, y \in \mathbb{Z}_p, \tag{1}$$

with the default prime $p = 97$. The project then extends the same framework to modular subtraction and division, polynomial expressions over $\mathbb{Z}_p$, operations in the symmetric group S_5, and K-ary modular summation. The goal is not only to reproduce a delayed generalization curve, but also to test which implementation and training choices make the phenomenon stronger or weaker.

2 Problem Formulation and Implementation

2.1 Sequence format

Every example is represented as a short token sequence. For binary modular tasks, the stored sample is

$$[x, \mathrm{op}, y, \mathrm{eq}, z], \tag{2}$$

where z is the target answer. During training, the model receives the prefix and a dummy equality token at the final position, and the loss is applied only to the final output distribution. This turns the problem into classification over the vocabulary while avoiding irrelevant language modeling losses on the input tokens.

For K-ary summation, the sequence is extended as

$$[x_1, \mathrm{op}, x_2, \mathrm{op}, \dots, \mathrm{op}, x_K, \mathrm{eq}, z], \qquad z = \sum_{i=1}^{K} x_i \bmod p. \tag{3}$$

In the implementation, this gives a sequence length of $2K + 1$. When K is large, the full Cartesian product has size p^K, so the data generator caps the sampled dataset size to avoid an immediate combinatorial explosion.

2.2 Datasets

The code in `mlfinal/data.py` constructs three families of datasets:

Table 1: Task families supported by the project.

Family	Operations
Modular arithmetic	`mod_add`, `mod_sub`, `mod_div`, parity based division or subtraction
Polynomial modular tasks	$x^2 + y^2$, $x^2 + xy + y^2$, $x^3 + xy$, $x^3 + xy^2 + y$, and variants
Group operations	multiplication, conjugation, and xyx in S_5
K-ary tasks	`sum_mod` and `mod_add` with $K > 2$

The data are shuffled using a fixed random seed and split according to a configurable training fraction α. This fraction is central to grokking: small α makes memorization easier to separate from true rule learning, while large α gives the model more direct coverage of the operation table.

2.3 Models

The main model is a decoder-only Transformer implemented in `mlfinal/architectures.py`. The default configuration uses two transformer blocks, hidden dimension 128, four attention heads, learned token embeddings, learned positional embeddings, causal self-attention, layer normalization, residual connections, GELU feedforward layers, and dropout.

The same training loop also supports three comparison architectures:

- an MLP that embeds each token, flattens the sequence, and predicts logits for every position;
- an LSTM with learned token and positional embeddings;
- a GRU with the same input representation as the LSTM.

These alternatives test whether grokking is a Transformer-specific behavior or a broader property of learning finite algorithmic tables with gradient descent.

2.4 Training loop

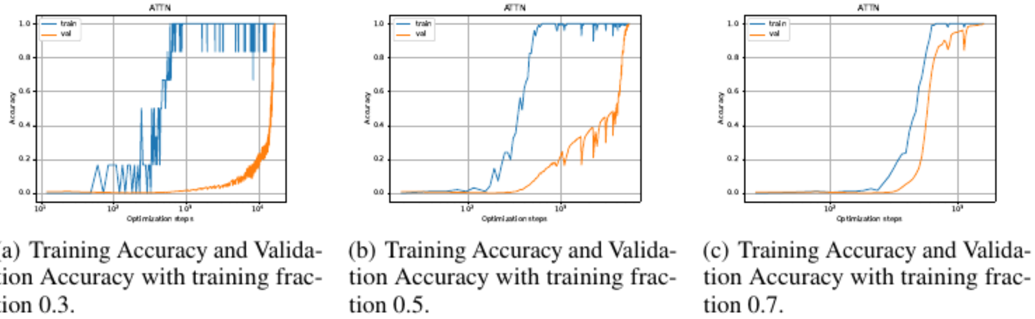
The training loop in `mlfinal/trainer.py` uses cross-entropy loss on the final token. The default optimizer is AdamW with learning rate 10^{-3}, weight decay 1.0, linear warmup, and gradient clipping. The implementation also supports Adam, SGD with momentum, and RMSprop. Additional controls include dropout, Gaussian gradient noise, Gaussian weight noise, full-batch training, custom evaluation intervals, early stopping after reaching a target validation accuracy, and checkpointing of the best validation model.

Each run writes a configuration file, a CSV training log, optional accuracy plots, and a best-model checkpoint through `mlfinal/storage.py`. These logs make it possible to compare grokking onset across architectures, optimizers, and data fractions without changing the experiment code.

3 Experimental Observations

3.1 Effect of training data fraction

The clearest grokking curves occur when the training fraction is neither too small to learn nor so large that validation examples are effectively covered by neighboring training examples. Figure 1 shows representative curves from the project material for modular addition at training fractions 0.3, 0.5, and 0.7. In all three cases the training curve rises before the validation curve. The delay is most visible when the data fraction is lower, because a memorized table is less useful on held-out pairs.



(a) Training Accuracy and Validation Accuracy with training fraction 0.3.

(b) Training Accuracy and Validation Accuracy with training fraction 0.5.

(c) Training Accuracy and Validation Accuracy with training fraction 0.7.

Figure 1: Representative training and validation accuracy curves for modular addition with different training fractions. The blue curve is training accuracy and the orange curve is validation accuracy. Lower training fractions make the memorization to generalization gap more visible.

This behavior supports the core interpretation of grokking. Early training finds a high-capacity solution that fits the observed pairs. Continued optimization, especially with weight decay, gradually favors a lower-complexity solution that captures the modular rule and therefore transfers to unseen pairs.

3.2 Architecture comparison

The project compares Transformers, MLPs, LSTMs, and GRUs through a shared command line interface. The important observation is that the delayed generalization pattern is not exclusive to self-attention. A Transformer has a natural sequence modeling bias and often reaches strong validation accuracy efficiently, but the MLP and recurrent models can also show a separation between memorization and generalization. This suggests that grokking is driven less by a single architectural trick and more by the interaction between algebraic structure, optimization, and regularization.

3.3 Optimizer and regularization effects

Optimizer choice changes the time scale of grokking. AdamW is the most natural default in this project because decoupled weight decay directly penalizes large parameter norms while retaining adaptive updates. Standard Adam, RMSprop, and SGD can also be tested, but they typically require more careful tuning of learning rate, momentum, and weight decay.

Regularization has a particularly direct role. Weight decay encourages solutions that fit the training set with smaller parameter norm. Once memorization becomes less favorable under this pressure, the model can move toward a more systematic algorithmic representation. Dropout can improve data efficiency, while excessive noise can slow the transition by disturbing the optimization path. These effects match the view that grokking is not just "training longer", but training longer under biases that make the rule-based solution attractive.

3.4 Task complexity and K-ary summation

The codebase extends the binary modular task to harder operations. Modular division is more difficult than addition because the inverse structure must be learned. Polynomial tasks introduce higher-order interactions. The S_5 tasks replace integer residues with permutations and therefore test whether the model can learn nontrivial group composition. Finally, K-ary summation increases the sequence length and the number of possible inputs. As K grows, the data space grows as p^K, so a fixed number of samples covers a rapidly shrinking portion of the full table. This makes memorization less reliable and pushes the model toward learning the compositional addition rule.

4 Explaining Grokking

A useful way to explain grokking is to separate two solutions that both reduce training loss. The first solution is table memorization: the model stores enough idiosyncratic information to answer the observed pairs. This is easy when the dataset is small and the network is overparameterized. The second solution is rule learning: the model represents the underlying algebraic operation. This second solution is harder to find initially, but it is more compact and generalizes to held-out combinations.

Weight decay helps because it changes the relative attractiveness of these two solutions. A memorized table can fit the training data, but it may require a larger or less organized parameter configuration. A rule-based

solution can fit both training and validation data with a simpler internal representation. During the long plateau, validation accuracy can remain low even though training accuracy is high because the model has not yet moved from the first solution to the second. The sudden rise in validation accuracy marks the point where the learned representation becomes algorithmic enough to transfer.

For modular addition, commutativity also matters. If both (x, y) and (y, x) are split across train and validation, then a model may improve validation accuracy by learning symmetry before it learns the full modular rule. This can partially mask grokking. Datasets that reduce this shortcut, such as half-table constructions or splits with low commutative overlap, make the delayed transition clearer.

5 Reproducibility

The repository is designed to make new runs easy to reproduce. A quick smoke test is:

```
python train.py --op mod_add --steps 3000 --target-val-acc 0.90
```

Typical longer runs use a sparse training fraction and strong regularization:

```
python train.py --op mod_add --train-ratio 0.25 --steps 100000 \
  --target-val-acc 0.9995 --optimizer adamw --dropout 0.1
```

Architecture comparisons can be run by changing only the model argument:

```
python train.py --architecture transformer
python train.py --architecture mlp
python train.py --architecture lstm
python train.py --architecture gru
```

Harder task families can be selected with `--op`, for example `s5_mul` for group multiplication or `sum_mod --k 3` for ternary modular summation.

6 Conclusion

This project provides a compact experimental platform for studying delayed generalization. The implementation confirms the main qualitative pattern of grokking: training accuracy can saturate long before validation accuracy improves, and the final transition can be abrupt. The experiments and code indicate that this behavior is influenced by training data fraction, optimizer choice, regularization strength, model architecture, and algebraic task complexity. The most convincing explanation is that grokking reflects a change in the solution preferred by optimization: from memorizing a finite table toward representing a compact rule. Because the code supports multiple architectures and task families under one interface, it can serve as a useful base for further studies of when neural networks move from memorization to algorithmic generalization.

References

[1] Power, A., Burda, Y., Edwards, H., Babuschkin, I., and Misra, V. (2022). Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177.*

[2] Liu, Z., Kitouni, O., Nolte, N. S., Michaud, E. J., Tegmark, M., and Williams, M. (2022). Towards understanding grokking: An effective theory of representation learning. *Advances in Neural Information Processing Systems.*

[3] Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. *International Conference on Learning Representations.*

[4] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems.*