Beyond Memorization: Exploring the Dynamics of Grokking in Sparse Neural Networks

by

Siwakorn Fuangkawinsombut

S.B. Mathematics and Computer Science and Engineering,
Massachusetts Institute of Technology, 2023

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

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

© 2024 Siwakorn Fuangkawinsombut. This work is licensed under a CC BY-NC-ND 4.0 license.

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

Authored by: Siwakorn Fuangkawinsombut
Department of Electrical Engineering and Computer Science
May 17, 2024

Certified by: Srinivasan Raghuraman
Lecturer of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Beyond Memorization: Exploring the Dynamics of Grokking in Sparse Neural Networks

by

Siwakorn Fuangkawinsombut

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In the domain of machine learning, "grokking" is a phenomenon where neural network models demonstrate a sudden improvement in generalization, distinct from traditional learning phases, long after the initial training appears complete. This behavior was first identified by Power et al. (2022) [5]. This thesis explores grokking within the context of the (n, k)-parity problem, aiming to uncover the mechanisms that trigger such transitions. Through extensive empirical research, we examine how different neural network configurations and training conditions influence the onset of grokking. Our methodology integrates advanced visualization techniques, such as t-SNE, and kernel density estimations to track the evolution from memorization to generalization phases. Furthermore, we investigate the roles of weight decay and network robustness against outliers, focusing on optimizing neural network architectures to achieve effective generalization with fewer computational resources. This study advances our understanding of grokking and proposes practical strategies for designing more efficient neural networks.

Thesis supervisor: Srinivasan Raghuraman
Title: Lecturer of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I must express my deepest gratitude to my thesis supervisor, Srinivasan Raghuraman. His guidance on this research topic and willingness to explore new ideas have been fundamental to my growth. The semester spent under his mentorship for this project has been invaluable. His support has been crucial, particularly through the 6.1220 TA funding, which enabled me to pursue my studies at MIT. It was in his class, where he first taught me as a TA and where I now assist him as a TA, that I decided to major in computer science and continue towards a master's degree in this field. Without his unwavering support, this thesis would not have been possible.

I am also grateful to the instructors of 6.S898 for introducing me to the topic of grokking and to my final project teammate, who helped lay the groundwork for my thesis. I thank the Economics Department for the opportunity to teach both the foundational course 14.01 and the advanced course 14.19. These teaching roles have not only been immensely enriching but also supported my tuition for the first semester, without which this thesis would not have been completed.

My thanks also extend to my family, my girlfriend MM, and my friends from MIT'23—Arun and Krit. Additionally, my MIT friends Bom, Dhorn, Tiger, and Zoom, as well as my econ TA colleagues Andrew, Win, and David, and my high school friends from Thailand, Natch and Palm, have all been pillars of support. While there are many others who have contributed to my journey, space constraints prevent me from naming all, but I hold deep appreciation for each one.

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 9

List of Tables 11

1 Introduction 13
- 1.1 Understanding Grokking . . . 13
- 1.2 Significance and Motivation . . . 13
- 1.3 Literature Review . . . 14
- 1.4 Research Objectives . . . 14
- 1.5 Preliminary Findings . . . 15
- 1.6 Thesis Outline . . . 16

2 Investigation of Two Circuits 17
- 2.1 Methodology . . . 17
- 2.2 Visualization and Analysis . . . 18
  - 2.2.1 t-SNE Visualization . . . 18
  - 2.2.2 Kernel Density Estimation (KDE) . . . 19
  - 2.2.3 Spectral Norm Analysis . . . 19
- 2.3 Results and Discussion . . . 20

3 Investigation of Learning Dynamics 21
- 3.1 Influence of Weight Decay . . . 21
- 3.2 Robustness Against Outliers . . . 23
  - 3.2.1 Adjusting Weight Decay to Enhance Robustness . . . 24

4 Optimizing Neural Networks for (n, k)-Parity 27
- 4.1 Foundational Principles of Parity Computation in Neural Networks . . . 27
- 4.2 Optimizing Network Architecture for $(n, 2)$-Parity . . . 28

4.2.1 Network Construction 29
4.2.2 Output Layer Configuration 29
4.2.3 Empirical Result 30
4.3 Optimizing Network Architecture for $(n, 3)$-Parity 31
4.3.1 Experimental Approach and Outcomes 31
4.3.2 Future Research Directions 31

5 Conclusion **33**
5.1 Summary 33
5.2 Limitations 34
5.3 Future Work 34

A Extended Configuration Analysis **37**
A.1 Analysis for $k = 3$ 37

B Code Listing **41**
B.1 Data Preparation 41
B.2 1-ReLU Model and Hinge Loss 42
B.3 Utility Functions for Norms and Activations 43
B.4 Plotting Functions 44
B.5 Train Model 49

References **55**

List of Figures

1.1 Loss and accuracy of 1-layer ReLU networks for $(50, 2)$-parity 15

2.1 Loss and accuracy dynamics of the 1-layer ReLU network for $(50, 2)$-parity, showcasing epochs indicative of grokking phenomena. 18
2.2 Dynamic t-SNE visualization of learned representations across epochs, highlighting the transition from memorization to effective generalization. 19
2.3 KDE analysis of the network's weights over various epochs, showing a trend towards sparsity. 19
2.4 Spectral norm progression of the weight matrix over epochs, aligned with key phases of learning. 20

3.1 Accuracy plots demonstrating the impact of varying weight decay on model generalization. 22
3.2 Spectral norm analysis showing convergence rates under different weight decay settings. 22
3.3 Model accuracy over epochs with a low weight decay ($w = 0.01$). Grokking still occurs when $p_t = 0.02$, but generalization efficacy is reduced as p_t increases. . . . 23
3.4 Model accuracy over epochs with a low weight decay ($w = 0.01$). Grokking does not occur for $p_t \geq 0.05$, demonstrating the model's decreased ability to generalize as outlier proportion increases. 24
3.5 Model accuracy with increased weight decay ($w = 0.1$) as outlier fraction varies, indicating some capability to overcome outliers. 25

A.1 Loss and accuracy dynamics of the 1-layer ReLU network for $(50, 3)$-parity, showcasing epochs indicative of grokking phenomena. 37
A.2 Spectral norm dynamics across epochs. 38
A.3 Dynamic t-SNE visualization across epochs. 38
A.4 KDE of the network's weights over epochs. 39

List of Tables

3.1 Threshold values (p_t) for "tricked" configurations where grokking still occurs, categorized by number of input bits and weight decay settings. 25

Chapter 1

Introduction

1.1 Understanding Grokking

The phenomenon of "grokking," first observed and named by Power et al. (2022) [5], represents a fascinating and sudden transition in machine learning where models exhibit a leap from memorization to profound generalization ability. This phenomenon occurs unexpectedly, often after a model appears to have stabilized its performance on training data. The term itself, derived from the science fiction novel "Stranger in a Strange Land" by Robert A. Heinlein, implies a deep, intuitive understanding—mirroring the sudden enhancement in model performance that goes beyond traditional learning curves. This dissertation investigates the grokking phenomenon within the framework of neural networks trained on the (n, k)-parity problem, a well-defined mathematical challenge that offers clear metrics for analysis and interpretation.

1.2 Significance and Motivation

The discovery of grokking has sparked significant interest in the machine learning community due to its implications for the development of learning algorithms that can achieve true generalization. Traditional learning models often rely on incremental improvements during training; however, grokking suggests a potential for models to switch abruptly to a vastly more effective generalization

regime. This has profound implications for our understanding of how neural networks learn and might suggest new pathways to creating more efficient and powerful AI systems.

1.3 Literature Review

Since its initial observation, several theories have been proposed to explain the mechanisms behind grokking. Noteworthy among them are the studies by Liu et al. (2023) [3], which suggest that specific parameter initializations at scale can predispose models to grokking, and Thilak et al. (2022) [6], which focus on the role of optimizer dynamics in triggering these learning phenomena. More comprehensive theoretical frameworks, such as those proposed by Barak et al. (2022) [1], hypothesize that grokking emerges from a gradual development of structured and effective internal representations.

Further research has refined these ideas, exploring the subtleties of network architecture and training conditions that might influence or predict the occurrence of grokking. Recent contributions in this area, including those by Davies et al. (2023) [2] and Merrill et al. (2023) [4], have examined the impacts of inductive biases and the competition between different network substructures during the training process.

1.4 Research Objectives

This research aims to systematically dissect the phenomenon of grokking within the controlled context of the (n, k)-parity problem. By manipulating network architectures and training parameters, this study seeks to:

- Identify the conditions under which grokking occurs.
- Understand the role of network topology and training dynamics in inducing grokking.
- Explore the potential for reducing the complexity of network models while maintaining or enhancing their generalization capabilities.

1.5 Preliminary Findings

Our initial experiments have focused on simple configurations of the (n, k)-parity problem, particularly for $k = 2$ and $k = 3$. These experiments have provided insights into how minimal network architectures can be optimized to exhibit grokking, shedding light on the balance between network simplicity and learning effectiveness.

- Experimental Setup: We utilized 1-layer ReLU networks with a modest number of neurons, trained on datasets crafted to evaluate parity functions. These initial tests were instrumental in confirming the theoretical potential for grokking identified in prior studies.

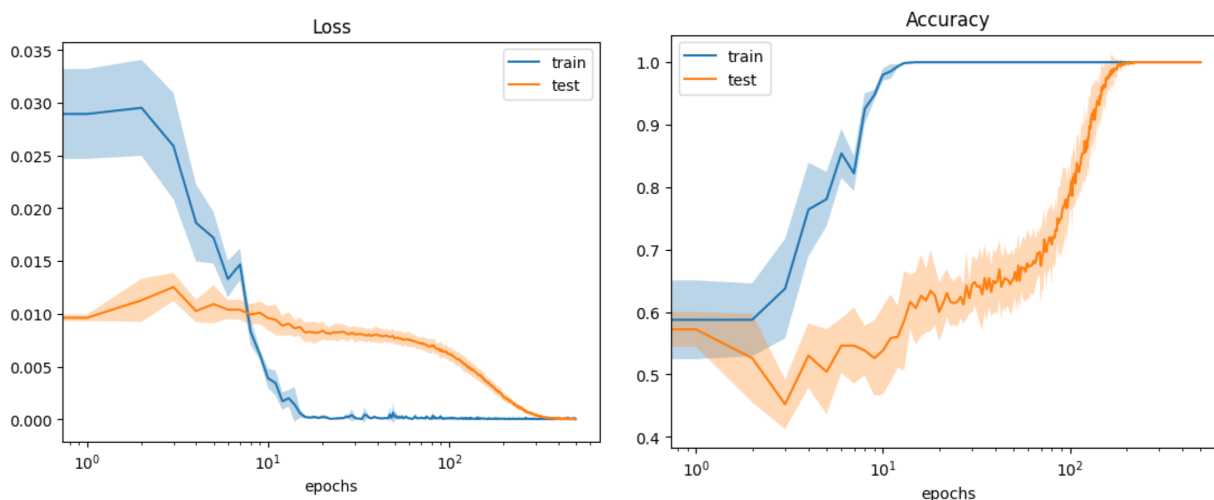


Figure 1.1: Loss and accuracy of 1-layer ReLU networks for $(50, 2)$-parity, averaged over five runs.

- Results: The networks demonstrated significant improvements in generalization ability, particularly after specific training milestones, consistent with the characteristics of grokking as described in the literature.

1.6 Thesis Outline

This introduction sets the stage for a comprehensive study of grokking, aimed at unraveling this complex phenomenon through a focused investigation of the (n, k)-parity problem. By integrating theoretical insights with empirical research, this thesis aims to contribute significantly to our understanding of how neural networks can achieve sudden leaps in generalization, pushing the boundaries of machine learning research.

Chapter 2

Investigation of Two Circuits

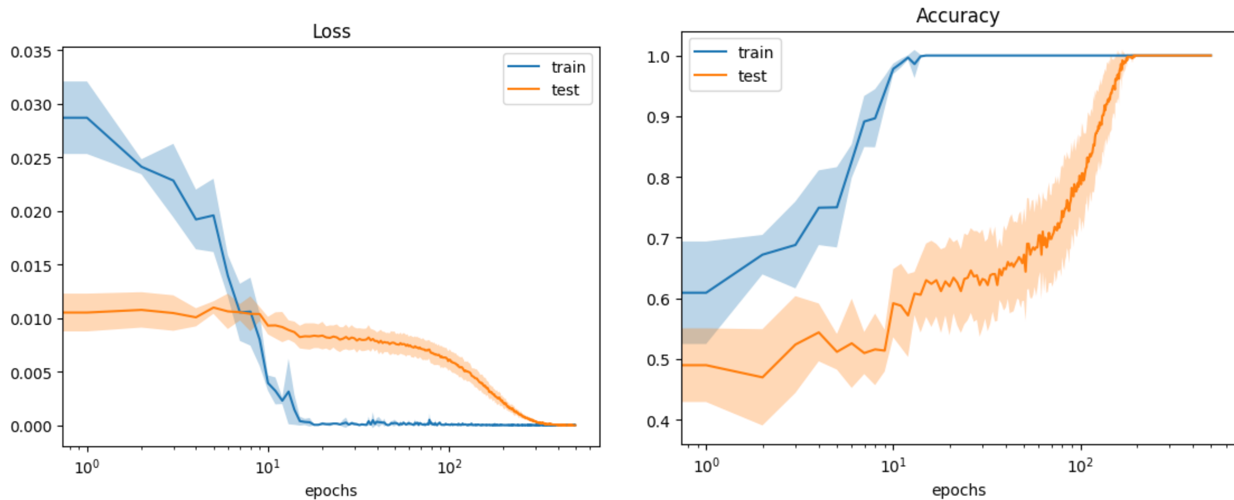
2.1 Methodology

Our study employs the (n, k)-parity problem, previously explored by Barak et al. (2022) [1] and Merrill et al. (2023) [4], as a foundational task for analyzing grokking in learning sparse (n, k)-parity functions. Defined as sparse when $k \ll n$, our experiments utilize configurations with $n = 50$, $k = 2$, a training set size of $N = 300$, and a neural network layer configured with $m = 500$ neurons.

We utilize a 1-layer ReLU network, defined as:

$$f(x) = u^T \sigma(Wx + b),$$

where $\sigma(x) = \max\{0, x\}$ is the ReLU activation, $u \in \mathbb{R}^m$, $W \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. Classification is determined by the sign of $f(x)$, predicting positive or negative class labels based on the parity output. The network minimizes the hinge loss $l(x, y) = \max\{0, 1 - f(x)y\}$, using stochastic gradient descent with a batch size of $B = 32$, a constant learning rate $\eta = 0.1$, and a weight decay parameter $\lambda = 0.01$.



(a) Training and testing loss across epochs.

(b) Training and testing accuracy across epochs.

Figure 2.1: Loss and accuracy dynamics of the 1-layer ReLU network for $(50, 2)$-parity, showcasing epochs indicative of grokking phenomena.

2.2 Visualization and Analysis

To empirically demonstrate grokking and support the theoretical concepts, we undertake a series of visualizations:

2.2.1 t-SNE Visualization

We examine the transformation of internal representations across various epochs, specifically looking at epochs where the model transitions from memorization to generalization. Our t-SNE plots color data according to the first $k = 2$ coordinates, resulting in $2^k = 4$ distinct colors. These visualizations reveal the emergence of "good" representations, signifying effective generalization, contrasting sharply with earlier "bad" representations that fail to cluster effectively.

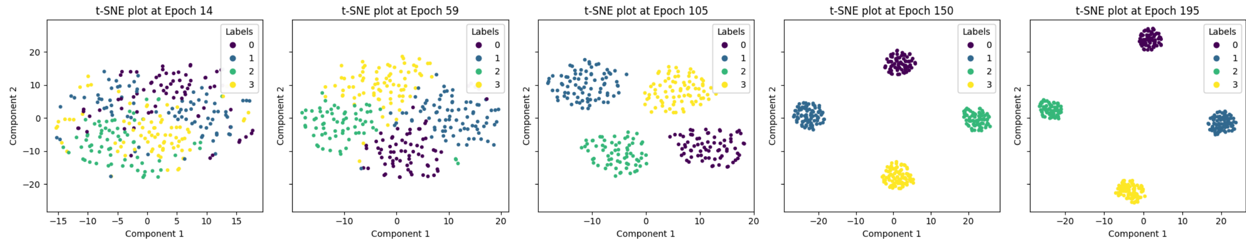


Figure 2.2: Dynamic t-SNE visualization of learned representations across epochs, highlighting the transition from memorization to effective generalization.

2.2.2 Kernel Density Estimation (KDE)

We analyze the distribution of weights within the network's hidden layer at different training stages. The KDE plots illustrate a significant condensation of weight values around zero as the network evolves, indicating a move towards a sparser and potentially more efficient configuration.

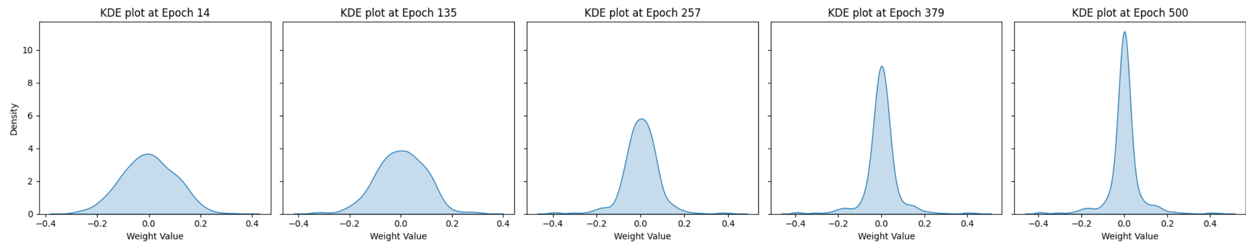


Figure 2.3: KDE analysis of the network's weights over various epochs, showing a trend towards sparsity.

2.2.3 Spectral Norm Analysis

The spectral norm of the weight matrix serves as a quantitative progress measure. Our analysis identifies two critical inflection points in the spectral norm corresponding to the epochs of memorization and generalization. This measure starkly declines between these points, underscoring a pivotal shift in network behavior.

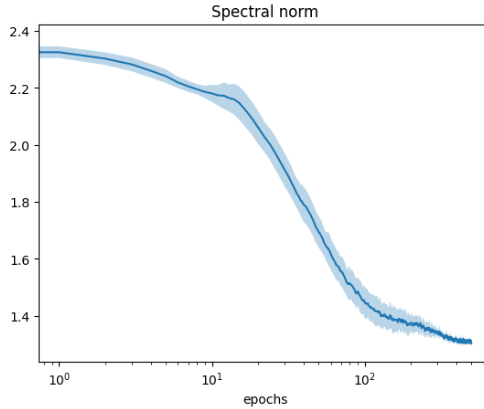


Figure 2.4: Spectral norm progression of the weight matrix over epochs, aligned with key phases of learning.

2.3 Results and Discussion

Our findings support the dual-circuit hypothesis, showing distinct phases of memorization and generalization within the training process. These results, underpinned by various visualizations and spectral norm analysis, validate theoretical models and offer deeper insights into the neural dynamics underpinning grokking. The comprehensive analysis highlights the complex interplay between network architecture, learning dynamics, and task complexity essential for effective neural network generalization. For a more detailed exploration of the model's performance on the more complex $k = 3$ case, including extended epoch training and additional diagnostic plots, see the extended configuration analysis in the appendix $A.1$. This section delves deeper into how extended training periods influence learning dynamics and generalization in higher-dimensional parity tasks.

Chapter 3

Investigation of Learning Dynamics

Observations from our study suggest the presence of both memorizing and generalizing circuits within neural networks. Understanding how networks transition from memorization to generalization—termed grokking—is crucial for explaining complex learning behaviors in artificial neural systems. This chapter explores various mechanisms that potentially facilitate this transition, focusing particularly on the role of weight decay, as influenced by prior work from Power et al. (2022) [5] and Varma et al. (2023) [7].

3.1 Influence of Weight Decay

Weight decay is hypothesized to play a critical role in promoting grokking by penalizing large weights, thus helping to sparsify the network and facilitate a shift from memorizing to generalizing circuits. Varma et al. (2023) [7] emphasize that while weight decay is typically known for its regularization effects, in the context of grokking, it appears to accelerate the onset of generalization by modifying the network's learning trajectory.

Hypothesis 3.1: Weight Decay Hypothesis

Increasing the weight decay parameter λ in the optimizer after the model has achieved memorization will accelerate generalization. However, there exists a threshold of weight decay beyond which the model fails to generalize.

This hypothesis aligns with suggestions from Varma et al. (2023) [7] and is tested by varying the weight decay λ post-memorization. As demonstrated in Figure 3.1, as weight decay increases, the model's ability to generalize improves rapidly. However, beyond a critical threshold, approximately between $\lambda = 0.1$ and $\lambda = 0.5$, the beneficial effects of weight decay diminish, and generalization capabilities deteriorate.

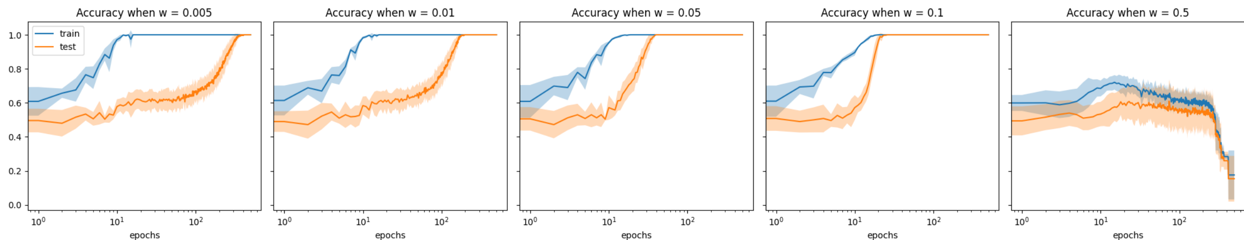


Figure 3.1: Accuracy plots demonstrating the impact of varying weight decay on model generalization.

Moreover, the spectral norms of the weight matrices, shown in Figure 3.2, further elucidate this phenomenon. Appropriate levels of weight decay support the network in converging towards the spectral norms characteristic of the final generalizing circuit. However, excessively high weight decay leads to a rapid convergence of the spectral norm to zero, indicating a loss of capacity to maintain and utilize learned features effectively.

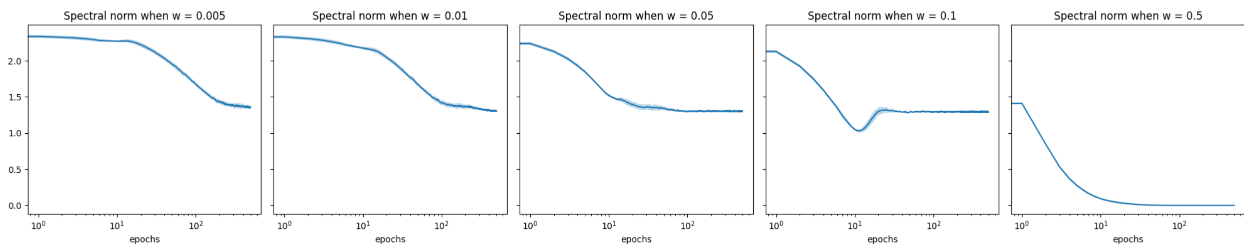


Figure 3.2: Spectral norm analysis showing convergence rates under different weight decay settings.

These findings support the Weight Decay Hypothesis 3.1 by illustrating that while moderate weight decay can facilitate faster transition from memorization to generalization by reducing overfitting and promoting robust feature learning, too much decay can strip the network of its ability to learn and generalize from the training data effectively.

3.2 Robustness Against Outliers

Another aspect of our investigation concerns the model's ability to handle outliers, a test for the robustness of the learned-representation theory of grokking. Understanding a model's robustness against outliers is critical for assessing its generalization capabilities in real-world scenarios. Outliers in the training data can significantly affect the learning process, potentially leading to overfitting or underfitting, depending on the model's ability to identify and disregard these anomalies. This section explores how increasing the proportion of outliers impacts the learning dynamics of neural networks, particularly their ability to transition from memorization to generalization.

Hypothesis 3.2: Outlier Hypothesis

As the proportion of outliers p_t in the training data increases, the speed of generalization decreases, eventually reaching a threshold beyond which the model fails to generalize.

In our experiments, p_t represents the fraction of training data labels that are randomly flipped, introducing significant noise into the dataset. This setup aims to simulate real-world scenarios where data may be corrupted or misleading, challenging the model's ability to generalize effectively.

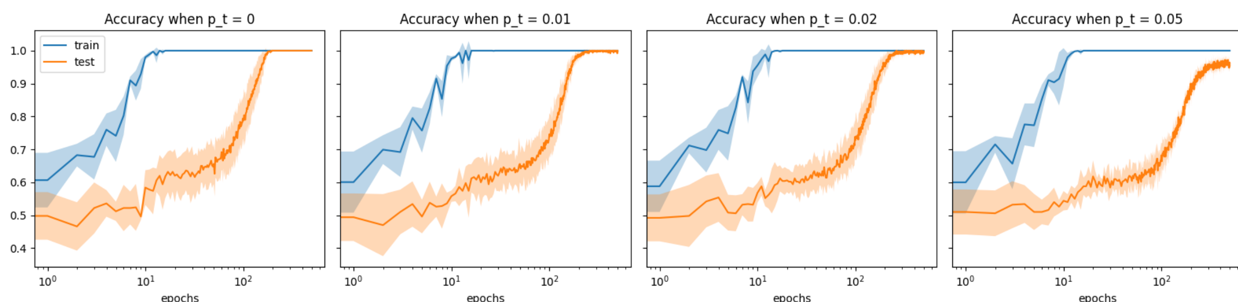


Figure 3.3: Model accuracy over epochs with a low weight decay ($w = 0.01$). Grokking still occurs when $p_t = 0.02$, but generalization efficacy is reduced as p_t increases.

Initially, with a weight decay setting of $w = 0.01$, we observed that the model struggles to maintain high generalization accuracy as p_t increases. Notably, when p_t reaches 0.02, the model still exhibits some ability to generalize, albeit at a reduced efficacy. However, for p_t values of 0.05 and beyond, the model fails to generalize effectively, as illustrated in Figure 3.3.

The observation that grokking does not occur when the fraction of outliers reaches 0.05 and beyond highlights a critical threshold for the model's ability to handle label noise. This phenomenon suggests a potential avenue for adjusting model parameters, such as weight decay, to enhance generalization in the presence of outliers. This hypothesis is further explored by increasing the weight decay to $w = 0.1$ to determine if it can mitigate the negative effects of higher p_t values on the model's generalization ability.

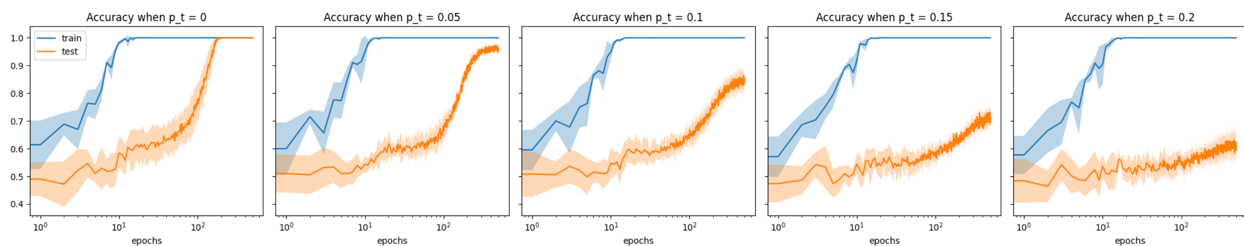


Figure 3.4: Model accuracy over epochs with a low weight decay ($w = 0.01$). Grokking does not occur for $p_t \geq 0.05$, demonstrating the model's decreased ability to generalize as outlier proportion increases.

3.2.1 Adjusting Weight Decay to Enhance Robustness

Given the deterioration in generalization performance with $p_t \geq 0.05$ under lower weight decay, we hypothesized that increasing the weight decay might bolster the model's resilience against outliers. Thus, we adjusted the weight decay to $w = 0.1$ and re-evaluated the model's performance. This adjustment revealed that the model could extend its generalization capabilities to $p_t = 0.1$, but not beyond $p_t = 0.15$, suggesting a nuanced interplay between weight decay and outlier resilience.

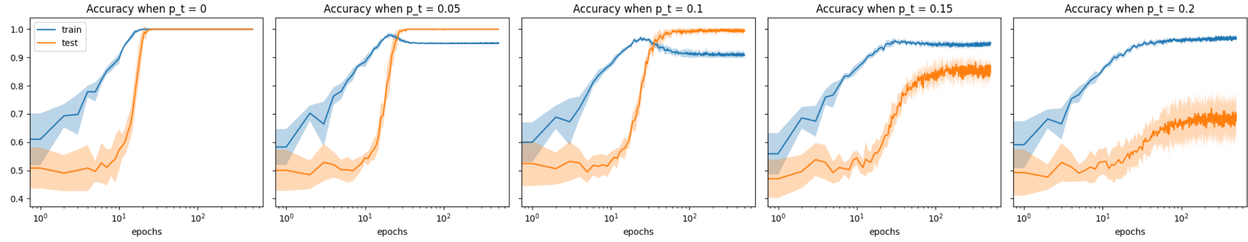


Figure 3.5: Model accuracy with increased weight decay ($w = 0.1$) as outlier fraction varies, indicating some capability to overcome outliers.

These findings support our hypothesis that increasing the weight decay parameter can mitigate some negative effects of outliers by promoting a sparser and potentially more robust representation, which in turn aids generalization. However, the effectiveness of this strategy has its limits, emphasizing the need for careful tuning of model parameters in environments with high data variability.

To further delineate the boundary conditions for successful generalization, we expanded our investigation to include variations in the number of input bits (n) and different levels of weight decay. We conducted experiments with three distinct settings of weight decay: 0.01, 0.05, and 0.1, alongside adjustments to the number of input bits, reducing to $n = 30$ and increasing to $n = 70$ from the standard $n = 50$. This exploratory analysis aimed to discern the interplay between network size, weight decay, and their combined effect on the model's capacity to handle outliers.

Table 3.1: Threshold values (p_t) for "tricked" configurations where grokking still occurs, categorized by number of input bits and weight decay settings.

Number of Input Bits (n)	Weight Decay (w)		
	0.01	0.05	0.1
30	0.02	0.12	0.20
50	0.03	0.06	0.12
70	0.01	0.03	0.05

Analysis of the data from this table reveals that as weight decay increases, the threshold p_t for maintaining grokking also rises. Conversely, increasing the number of input bits generally lowers the threshold, indicating a decreased tolerance to outliers. Interestingly, while an increase in threshold is noted from $n = 30$ to $n = 50$ at $w = 0.01$, further increasing the number of input bits to $n = 70$ reverses this trend, suggesting a complex dynamic between the number of input bits and the ability to generalize in the presence of noise. These observations underscore the critical balance required between the number of input bits, weight decay, and the proportion of outliers to optimize neural network performance and generalization across different settings. Future research should explore the boundaries of this balance further, aiming to develop guidelines for optimal parameter settings in various operational contexts.

Chapter 4

Optimizing Neural Networks for (n, k)-Parity

This chapter explores constructing and optimizing neural networks to compute the (n, k)-parity function with minimal neuron usage. The (n, k)-parity problem presents a unique challenge for neural computation, as it requires the network to learn highly specific and granular patterns across various input combinations.

Inspired by the recent advances in neural architectures for parity functions by Merrill et al. (2023) [4], we aim to extend these methods to achieve more efficient network structures. These researchers demonstrated that a 1-layer ReLU network could effectively solve the $(n, 3)$-parity problem using only six neurons. We propose to investigate whether similar or better efficiency can be realized for different parity sizes, starting with $k = 2$ and progressing to $k = 3$.

4.1 Foundational Principles of Parity Computation in Neural Networks

Before delving into the details of optimized neural architectures, it's essential to establish a fundamental understanding of how neural networks can encode and compute parity functions. Establishing this baseline is critical as it sets the stage for exploring more efficient architectures that potentially use fewer neurons than theoretically suggested by the initial model.

Preposition 4.1 *For any n, there exists a 1-layer ReLU network with 2^k neurons that computes the (n,k)-parity function. [4]*

Proof. The proposed network consists of 2^k neurons, where each neuron is responsible for one of the 2^k possible combinations of k input bits from an n-dimensional vector. The neurons are configured such that each one "fires" (i.e., outputs a positive value) if and only if the subset of inputs it monitors has a product that equals 1, indicating a parity of 1.

This is achieved by setting the weights and biases of each neuron such that its activation function, a ReLU, returns a positive output for the desired combination and zero otherwise. By carefully setting the weights to positive or negative based on the input bits' expected contribution to the product (and adjusting the bias to offset the sum appropriately), the neuron will only activate for its designated input combination.

Thus, the network directly maps each input vector's parity conditions onto the activation patterns of these 2^k neurons. The final output of the network can be computed as the sign of the sum of all activated neurons' outputs. If any neuron indicating a parity of 1 is activated, the network's output will be positive (1); otherwise, it will be negative (-1), corresponding to no neurons firing or a parity of -1 across all monitored combinations.

This proof establishes that 2^k neurons are sufficient to represent and compute (n,k)-parity by using their activation states to encode each possible parity outcome directly. However, it raises the potential for optimizing the network by reducing the number of neurons if redundant or overlapping input conditions can be efficiently managed.

4.2 Optimizing Network Architecture for $(n,2)$-Parity

In the quest to optimize neural network architectures for the computation of parity functions, we begin by exploring the simplest non-trivial case, the $(n,2)$-parity problem. For this problem, we attempt to construct a neural network that uses only two neurons, the theoretical minimum for this task, to effectively compute parity.

The $(n, 2)$-parity function is computed based on the relationship between two designated input bits, x_1 and x_2. The goal is to determine whether the product of these two bits is 1 or -1, which corresponds to their parity being positive or negative, respectively. To achieve this with a neural network using ReLU activations, we design a minimal architecture that can directly capture this relationship.

4.2.1 Network Construction

We propose a network with two neurons, where each neuron is designed to activate based on the sum of the inputs. The neurons are defined as follows:

$$h_1 = \sigma(0.5x_1 + 0.5x_2)$$
$$h_2 = \sigma(-0.5x_1 - 0.5x_2)$$

Here, the coefficients 0.5 ensure that the contributions of x_1 and x_2 are balanced and that the activation reflects their combined effect.

4.2.2 Output Layer Configuration

In the final layer, we use a simple aggregation mechanism where both neurons contribute equally to the output:

$$y = h_1 + h_2$$

This configuration allows the network to output a value based on the activation of either h_1 or h_2. Since each neuron activates under specific conditions reflective of the input parity:

- h_1 activates when x_1 and x_2 are both positive, which corresponds to a positive product or a parity of 1.

- h_2 activates when x_1 and x_2 are both negative, which corresponds to a positive product or a parity of 1.

The network's output, y, effectively sums the outputs of h_1 and h_2. Given the nature of ReLU and the network design, this sum will be positive if the parity is positive and zero otherwise, correctly encoding the parity of x_1 and x_2.

4.2.3 Empirical Result

In our theoretical exploration, we demonstrated that a neural network with two neurons can effectively compute the $(n,2)$-parity function. This concept was not only validated theoretically but also confirmed through empirical training of our model. During the generalization phase, the model converged to a weight configuration that reflects the theoretical insights:

$$\text{First Layer Weights: } \begin{bmatrix} 0.62 & 0.62 \\ -0.62 & -0.62 \end{bmatrix}, \quad \text{Output Layer Weights: } [0.85, 0.85].$$

This weight matrix for the first layer effectively captures the essence of the $(n,2)$-parity problem, mirroring the theoretical model where each neuron is responsible for one configuration of the parity bits. The values 0.62 slightly exceed the expected 0.5, enhancing the model's sensitivity and robustness in recognizing parity, while the negative weights manage the opposite parity conditions.

The output layer's uniform weights of 0.85 reinforce the decisions made by the first layer, ensuring that the overall network output strongly represents the computed parity. This setup provides a clear empirical instance where the theoretical predictions not only hold but also exhibit practical viability through training dynamics.

This alignment between theory and practice underscores the effectiveness of the proposed network design and validates our initial hypothesis about the minimum neuron requirement for computing $(n,2)$-parity, highlighting the potential for further reducing neuron count in more complex parity computations.

4.3 Optimizing Network Architecture for $(n,3)$-Parity

Recent work by Merrill et al. (2023) [4] demonstrated the feasibility of solving the $(n,3)$-parity problem using a neural network architecture comprising only six neurons. Their theoretical analysis further suggested the potential to reduce the network complexity to just four neurons. However, these results were achieved under idealized theoretical conditions rather than through empirical grokking phenomena. Motivated by this gap, our research aimed to explore whether a network with five neurons could be effectively trained to generalize the $(n,3)$-parity function through grokking, representing a middle ground between empirical achievability and theoretical minimalism.

4.3.1 Experimental Approach and Outcomes

Our experimental design involved adjusting network parameters and training conditions to optimize for both sparsity and generalization. We utilized a hinge loss function, renowned for its efficacy in classification tasks by promoting margin maximization between classes. However, despite these efforts, the results deviated from our expectations: while the weights of the network approached zero, indicating a move towards sparsity, they did not fully reach zero. This outcome suggests an incomplete transition towards the ideal sparse configuration envisaged in theoretical models.

The inability to achieve a network configuration with exactly zero weights in some neurons raises significant questions about the limits of current training methodologies and loss functions in achieving theoretical minimalism in network design. This observation is particularly crucial for grokking, where the stark transition from memorization to generalization often hinges on subtle shifts in network parameters.

4.3.2 Future Research Directions

This experiment, while not successful in achieving its initial goal, provides valuable insights into the complex interplay between network architecture, training parameters, and loss functions in the context of grokking. Future research could explore:

- **Adjusting Training Size and Epochs:** Increasing the size of the training set or the number of training epochs could provide the network more opportunities to refine its weights and potentially achieve the desired sparsity.

- **Exploring Different Loss Functions:** Implementing loss functions that more rigorously enforce sparsity, such as L1 regularization, could help in pushing the weights to exact zeros.

- **Hybrid Training Approaches:** Combining traditional training methodologies with novel techniques such as meta-learning or transfer learning might enhance the network's ability to generalize from fewer examples, potentially enabling the achievement of sparsity with fewer neurons.

Chapter 5

Conclusion

This thesis explores the phenomenon of grokking within neural networks, particularly how these networks transition from memorization to a deeper, generalization phase. This exploration is rooted in the investigation of neural dynamics through the lens of the (n, k)-parity problem, revealing intricate behaviors and characteristics of learning dynamics that are critical for effective generalization. This final chapter summarizes the main points, discusses the limitations of the study, and outlines potential future work to further enhance and refine the grokking phenomenon.

5.1 Summary

The key contributions of this thesis are twofold. First, through rigorous experiments and analyses, we demonstrated how neural networks transition between memorization and generalization phases, a phenomenon we termed grokking. Our experiments with (n, k)-parity tasks, particularly for $k = 2$ and $k = 3$, provided empirical evidence supporting the dual-circuit hypothesis, which posits the existence of separate memorizing and generalizing circuits within trained neural networks.

Second, our research extended the practical understanding of the impact of weight decay and the handling of outliers on network performance. By varying weight decay parameters and introducing outliers into training data, we explored the robustness and adaptability of neural networks under diverse and challenging conditions. These experiments not only highlighted the delicate balance

required in tuning neural networks but also reinforced the importance of robust training regimes that can withstand real-world data anomalies.

5.2 Limitations

Despite the advancements made, this study faces several limitations:

- **Resource Limitations:** The computational demands of training larger models with higher k values or on more complex datasets were significant. This limitation was particularly acute given the resource-intensive nature of models capable of demonstrating grokking, which required extensive computational power not readily available to all researchers.
- **Nascent Field:** The concept of grokking, while promising, is still in its infancy. The theoretical underpinnings and mathematical frameworks remain underdeveloped, making it challenging to predict and manipulate grokking behaviors fully.
- **Empirical Focus:** Much of the research relied heavily on empirical observations which, while insightful, occasionally lacked the support of a strong theoretical backing that could unify the observed phenomena under a single explanatory framework.

5.3 Future Work

To build on the findings of this thesis, future research could take several directions:

- **Enhanced Computational Resources:** Investigating the feasibility of more complex models and larger datasets could be enabled by access to greater computational resources, potentially uncovering new dimensions of the grokking phenomenon.
- **Theoretical Advances:** Developing a more robust mathematical framework to describe and predict grokking could transform empirical observations into predictable outcomes, enhancing our ability to design neural networks that generalize effectively.

- **Alternative Architectures:** Exploring different network architectures and training paradigms, such as deep reinforcement learning or unsupervised learning models, could provide new insights into the conditions under which grokking occurs.

- **Exploration of $(n, 3)$-parity with Fewer Neurons:** Continuing to refine the approach to achieve a functioning model with fewer than six neurons for the $(n, 3)$-parity problem, possibly incorporating novel loss functions or training techniques to achieve theoretical predictions of network sparsity.

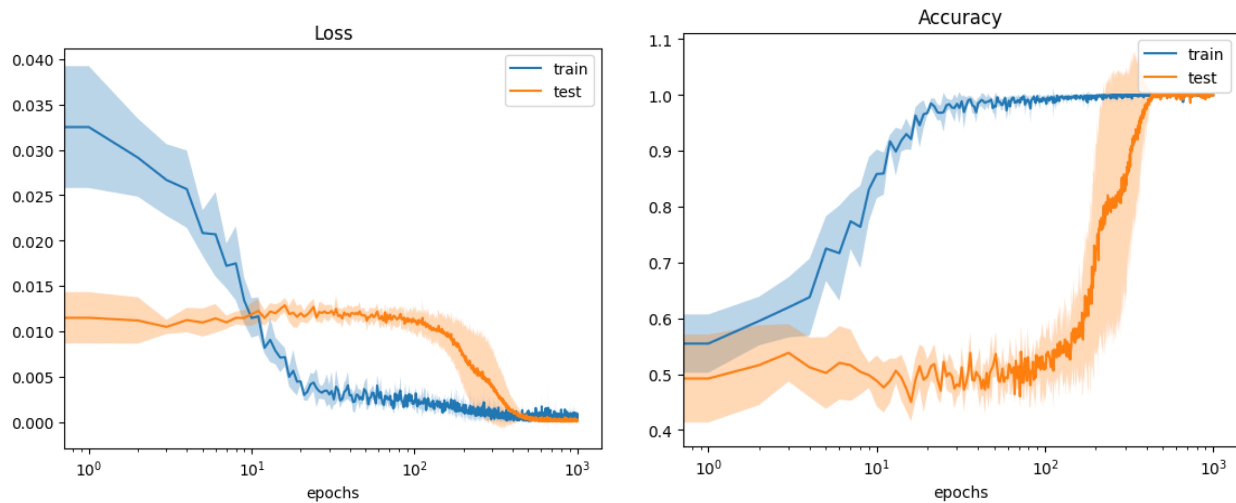
By addressing these limitations and pursuing suggested future work, the field can advance towards a more comprehensive understanding of how neural networks can be optimized for both performance and efficiency, moving closer to realizing the full potential of grokking in artificial intelligence.

Appendix A

Extended Configuration Analysis

A.1 Analysis for $k = 3$

For $k = 3$ case, we extended the training to 1000 epochs with a dataset size of 1000 to explore the robustness and performance of the neural network model. This section presents the results through detailed visual and statistical analyses, focusing on the grokking phenomenon and the evolution of network weights during training.



(a) Training and testing loss across epochs.

(b) Training and testing accuracy across epochs.

Figure A.1: Loss and accuracy dynamics of the 1-layer ReLU network for $(50, 3)$-parity, showcasing epochs indicative of grokking phenomena.

The loss plot shows a sustained decrease across epochs with the test loss eventually stabilizing after initial fluctuations, suggesting a successful adaptation and learning. In contrast, the accuracy plot demonstrates a significant improvement in test accuracy after a prolonged plateau phase, illustrating the network's ability to generalize, typical of the grokking phenomenon in more complex parity tasks.

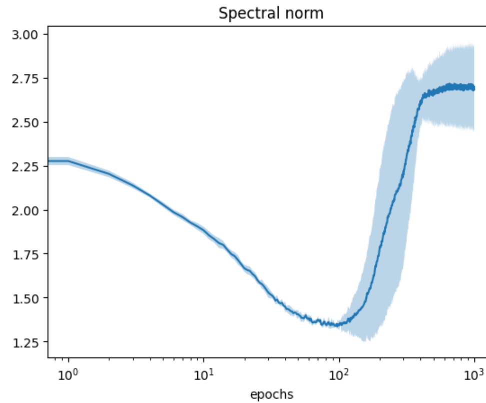


Figure A.2: Spectral norm dynamics across epochs.

The spectral norm plot reveals a notable increase in the spectral norm, correlating with the network's transition from memorization to effective generalization, as indicated by the stabilization of accuracy and loss in later epochs.

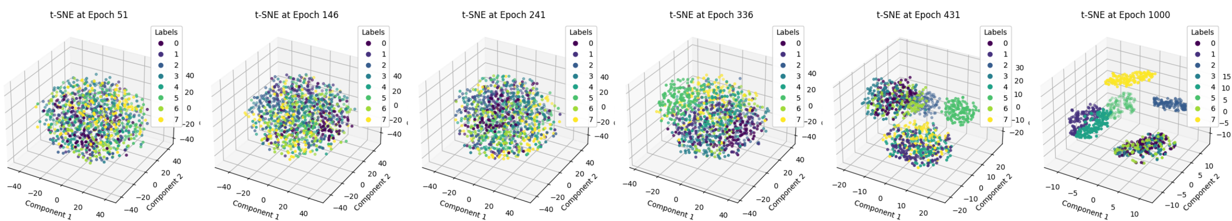


Figure A.3: Dynamic t-SNE visualization across epochs.

The t-SNE visualization across epochs shows clear clustering of labels in higher dimensions as epochs progress, reflecting how the network's internal representations evolve to effectively distinguish between different parity configurations.

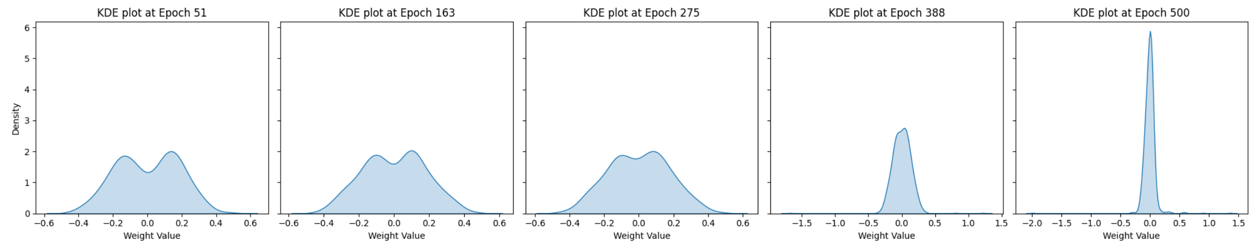


Figure A.4: KDE of the network's weights over epochs.

The KDE plot shows a trend towards weight sparsity and a narrowing peak over time, suggesting that the network is reducing redundant or less important weights as it optimizes for the $(50, 3)$-parity task. This indicates a refinement in the network's efficiency and computational focus.

This extended configuration analysis underlines the capability of 1-layer ReLU networks to adapt and refine their computational strategy for more complex parity problems, emphasizing the role of deep training and larger data sets in achieving nuanced generalization.

Appendix B

Code Listing

B.1 Data Preparation

```
def parity(n, k, n_samples, seed=898):
    random.seed(seed)
    samples = torch.Tensor([[random.choice([-1, 1]) for j in range(n)]
        for i in range(n_samples)])
    targets = torch.prod(samples[:, :k], dim=1)
    return samples, targets

def data_preparation(seed_id=898, test_samples=100, test_batchsize=100,
    return_raw=False):
    _data = parity(n, k, N, seed=seed_id)
    train_dataset = TensorDataset(_data[0], _data[1])
    train_dataloader = DataLoader(train_dataset, batch_size=B, shuffle=
        True)

    data = parity(n, k, test_samples, seed=2023)
    test_dataset = TensorDataset(data[0], data[1])
```

```
    test_dataloader = DataLoader(test_dataset, batch_size=
        test_batchsize, shuffle=True)

    if return_raw:
        return train_dataloader, train_dataloader, _data[0][:100], data
            [0]
    return train_dataloader, test_dataloader
```

B.2 1-ReLU Model and Hinge Loss

```
class FF1(nn.Module):
    def __init__(self, input_dim=50, width=1000):
        super(FF1, self).__init__()
        self.linear1 = nn.Linear(input_dim, width)
        self.activation = nn.ReLU()
        self.linear2 = nn.Linear(width, 1, bias=False)
        print(width)

    def forward(self, x, return_activation=False):
        x = self.linear1(x)
        x = self.activation(x)
        if return_activation:
            return x
        x = self.linear2(x)
        return x

    def print_weights(self):
        print("Weights of linear1:", self.linear1.weight.data)
```

```
        print("Bias of linear1:", self.linear1.bias.data)
        print("Weights of linear2:", self.linear2.weight.data)

class MyHingeLoss(nn.Module):
    def __init__(self):
        super(MyHingeLoss, self).__init__()

    def forward(self, output, target):
        hinge_loss = 1 - torch.mul(output.squeeze(), target.squeeze())
        hinge_loss = torch.clamp(hinge_loss, min=0)
        return hinge_loss.mean()
```

B.3 Utility Functions for Norms and Activations

```
def add_spectral_norm(norms, weight_matrix):
    svd_values = torch.linalg.svdvals(weight_matrix)
    spectral_norm = torch.max(svd_values).item()
    norms.append(spectral_norm)

def add_frobenius_norm(norms, weight_matrix):
    frobenius_norm = torch.linalg.matrix_norm(weight_matrix)
    norms.append(frobenius_norm)

def add_activations(activations, train_dataloader, model):
    epoch_activations = []
    for x_batch, _ in train_dataloader:
        x_batch = x_batch.to(device)
        with torch.no_grad():
```

```
            act = model(x_batch, return_activation=True).detach().cpu()
                .numpy()
        epoch_activations.append(act)
    activations.append(np.concatenate(epoch_activations, axis=0))

def add_labels(labels, train_dataloader, k):
    first_k_positions = []
    for x_batch, _ in train_dataloader:
        x_batch = x_batch.to(device)
        first_k_positions_batch = x_batch[:, :k].tolist()
        first_k_positions.extend(first_k_positions_batch)
    string_labels = [str(label) for label in first_k_positions]
    labels.append(string_labels)

def encode_labels(labels):
    unique_labels = sorted(set(labels))
    label_to_id = {label: idx for idx, label in enumerate(unique_labels
        )}
    return [label_to_id[label] for label in labels]
```

B.4 Plotting Functions

```
def plot_loss(losses, epochs, base_dir, log_scale=True, save=False):
    m1, std1 = mean_and_std_across_seeds(losses['train'])
    if save:
        np.save(os.path.join(base_dir, 'mean_train_loss'), m1)
        np.save(os.path.join(base_dir, 'std_train_loss'), std1)

```

```
    m2, std2 = mean_and_std_across_seeds(losses['test'])
    if save:
        np.save(os.path.join(base_dir, 'mean_test_loss'), m2)
        np.save(os.path.join(base_dir, 'std_test_loss'), std2)

    # Loss plot
    plt.plot(m1, linestyle='-', label='train')
    plt.plot(m2, linestyle='-', label='test')
    plt.fill_between([i for i in range(epochs)], m1 - std1, m1 + std1,
        alpha=0.3)
    plt.fill_between([i for i in range(epochs)], m2 - std2, m2 + std2,
        alpha=0.3)
    plt.title('Loss')
    plt.xlabel('epochs')
    if log_scale:
        plt.xscale('log')
    plt.legend()
    plt.show()

def plot_accuracy(accs, epochs, base_dir, log_scale=True, save=False):
    m1, std1 = mean_and_std_across_seeds(accs['train'])
    if save:
        np.save(os.path.join(base_dir, 'mean_train_acc'), m1)
        np.save(os.path.join(base_dir, 'std_train_acc'), std1)

    m2, std2 = mean_and_std_across_seeds(accs['test'])
    if save:
        np.save(os.path.join(base_dir, 'mean_test_acc'), m2)
        np.save(os.path.join(base_dir, 'std_test_acc'), std2)
```

```python

    # Accuracy plot
    plt.plot(m1, linestyle='-', label='train')
    plt.plot(m2, linestyle='-', label='test')
    plt.fill_between([i for i in range(epochs)], m1 - std1, m1 + std1,
        alpha=0.3)
    plt.fill_between([i for i in range(epochs)], m2 - std2, m2 + std2,
        alpha=0.3)
    plt.title('Accuracy')
    plt.xlabel('epochs')
    if log_scale:
        plt.xscale('log')
    plt.legend()
    plt.show()

def plot_norms(norms, epochs, log_scale=True):
    m, std = mean_and_std_across_seeds(norms)

    print(f"Final model's norm is {m[-1]}")

    plt.plot(m, linestyle='-')
    plt.fill_between([i for i in range(epochs)], m - std, m + std,
        alpha=0.3)
    plt.title('Spectral norm')
    plt.xlabel('epochs')
    if log_scale:
        plt.xscale('log')
    plt.show()

```

```
60 def plot_weights(weights, mem_epochs, gen_epochs, epochs, num_plots=5,
       include_last_epoch=True):
61     seed = max(range(len(gen_epochs)), key=lambda x: gen_epochs[x] -
           mem_epochs[x])
62     print(f"Weight plots for model at seed {seed}")
63     mem, gen = mem_epochs[seed], gen_epochs[seed]
64     gen = 499
65 
66     if gen - mem < 10:
67         gen = mem + 20
68 
69     # Correct number of plots based on include_last_epoch
70     total_plots = num_plots + 1 if include_last_epoch else num_plots
71     fig, axes = plt.subplots(1, total_plots, sharey=True, figsize=(
           total_plots * 4, 4))
72 
73     for idx, i in enumerate(np.round(np.linspace(mem, gen, num_plots)).
           astype(int)):
74         flattened_weights = weights[seed][i]
75         sns.kdeplot(flattened_weights, fill=True, ax=axes[idx])
76         axes[idx].set_title(f'KDE plot at Epoch {i + 1}')
77         axes[idx].set_xlabel('Weight Value')
78         axes[idx].set_ylabel('Density')
79 
80     if include_last_epoch:
81         flattened_weights = weights[seed][-1]
82         sns.kdeplot(flattened_weights, fill=True, ax=axes[num_plots])
83         axes[-1].set_title(f'KDE plot at Epoch {epochs}')
84         axes[-1].set_xlabel('Weight Value')
```

```
85          axes[-1].set_ylabel('Density')
86
87      plt.tight_layout()
88      plt.show()
89
90  def plot_tsne(epoch_activations, mem_epochs, gen_epochs, epochs, labels
        , num_plots=5, include_last_epoch=True):
91      seed = max(range(len(gen_epochs)), key=lambda x: gen_epochs[x] -
            mem_epochs[x])
92      print(f"t-SNE plots for model at seed {seed}")
93      mem, gen = mem_epochs[seed], gen_epochs[seed]
94
95      if gen - mem < 10:
96          gen = mem + 20
97
98      # Correct number of plots based on include_last_epoch
99      total_plots = num_plots + 1 if include_last_epoch else num_plots
100     fig, axes = plt.subplots(1, total_plots, sharey=True, figsize=(
            total_plots * 4, 4))
101
102     for idx, i in enumerate(np.round(np.linspace(mem, gen, num_plots)).
            astype(int)):
103         act = epoch_activations[seed][i]
104         label = encode_labels(labels[seed][i])
105         reduced_act = TSNE(n_components=2).fit_transform(act)
106         scatter = axes[idx].scatter(reduced_act[:, 0], reduced_act[:,
                1], c=label, s=10)
107         legend = axes[idx].legend(*scatter.legend_elements(), title="
                Labels", loc='upper right')
```

```
        axes[idx].add_artist(legend)
        axes[idx].set_title(f't-SNE plot at Epoch {i + 1}')
        axes[idx].set_xlabel('Component 1')
        axes[idx].set_ylabel('Component 2')

    if include_last_epoch:
        act = epoch_activations[seed][-1]
        label = encode_labels(labels[seed][-1])
        reduced_act = TSNE(n_components=2).fit_transform(act)
        scatter = axes[-1].scatter(reduced_act[:, 0], reduced_act[:,
            1], c=label, s=10)
        legend = axes[-1].legend(*scatter.legend_elements(), title="
            Labels", loc='upper right')
        axes[-1].add_artist(legend)
        axes[-1].set_title(f't-SNE plot at Epoch {epochs}')
        axes[-1].set_xlabel('Component 1')
        axes[-1].set_ylabel('Component 2')

    plt.tight_layout()
    plt.show()
```

B.5 Train Model

```
def train_model(seed_id, base_dir, save, model, epochs, loss_fn, losses
    , accs, norms, weights, epoch_activations, epoch_labels, mem_epochs,
    gen_epochs, k):
    torch.manual_seed(seed_id)

```

```
    # Data & save_dir preparation
    train_dataloader, test_dataloader = data_preparation(seed_id)
    path = os.path.join(base_dir, f'seed{seed_id}_checkpoints')
    os.makedirs(path, exist_ok=True)

    # Model & Optim initialization
    model = model.to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=lr, weight_decay
        =weight_decay)

    mem_epochs.append(-1)
    gen_epochs.append(-1)
    train_loss, test_loss = [], []
    train_acc, test_acc = [], []
    lin1_norms, lin2_norms = [], []
    cur_weights = []
    activations = []
    labels = []
    for epoch in range(epochs):
        if (epoch % 100 == 0):
            print(f'Epoch {epoch+1}/{epochs}, Train Acc: {acc_calc(
                train_dataloader, model):.4f}, Test Acc: {acc_calc(
                test_dataloader, model):.4f}')

        # Loss & Accuracy statistics
        train_loss.append(loss_calc(train_dataloader, model, loss_fn))
        test_loss.append(loss_calc(test_dataloader, model, loss_fn))

        train_acc.append(acc_calc(train_dataloader, model))
```

```python
        test_acc.append(acc_calc(test_dataloader, model))

        # Calculate and store the spectral norm
        with torch.no_grad():
            # First layer norm
            add_spectral_norm(lin1_norms, model.linear1.weight)
            # add_frobenius_norm(lin1_norms, model.linear1.weight)

            # Final layer norm
            add_spectral_norm(lin2_norms, model.linear2.weight)
            # add_frobenius_norm(lin2_norms, model.linear2.weight)

            flattened_weights = model.linear2.weight.detach().cpu().
               numpy().flatten()
            cur_weights.append(flattened_weights)

        # # Capture activations
        # add_activations(activations, train_dataloader, model)

        # Save memorizing / generalizing network
        if (train_acc[-1] > 0.999 and mem_epochs[-1] < 0):
            print(f'Saving memorizing model - epoch {epoch+1}')
            if save:
                torch.save(model.state_dict(), os.path.join(path, '
                   memorization.pt'))
            mem_epochs[-1] = epoch

        if (test_acc[-1] > 0.999 and gen_epochs[-1] < 0):
            print(f'Saving generalizing model - epoch {epoch+1}')
```

```
            gen_epochs[-1] = epoch

        # if (epoch == epochs - 1):
            # print(f'Saving (final) generalizing model - epoch {epoch+1}')
            # if save:
            #     torch.save(model.state_dict(), os.path.join(path, 'generalization.pt'))
            # print(f'Weights of generalizing model - epoch {epoch+1}')
            # model.print_weights()

        # Save model
        if save:
            torch.save(model.state_dict(), os.path.join(path, f'model_{epoch}.pt'))

        first_k_positions = []
        epoch_act = []
        # Train model
        for id_batch, (x_batch, y_batch) in enumerate(train_dataloader):
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
            with torch.no_grad():
                first_k_positions_batch = x_batch[:, :k].tolist()
                first_k_positions.extend(first_k_positions_batch)
                act = model(x_batch, return_activation=True).detach().cpu().numpy()
                epoch_act.append(act)

```

```
                pred = model(x_batch)

                optimizer.zero_grad()
                loss = loss_fn(pred, y_batch).mean()
                loss.backward()
                optimizer.step()

            string_labels = [str(label) for label in first_k_positions]
            labels.append(string_labels)
            activations.append(np.concatenate(epoch_act, axis=0))

        losses['train'].append(train_loss)
        losses['test'].append(test_loss)

        accs['train'].append(train_acc)
        accs['test'].append(test_acc)

        norms['lin1'].append(lin1_norms)
        norms['lin2'].append(lin2_norms)

        weights.append(cur_weights)

        epoch_activations.append(activations)
        epoch_labels.append(labels)
```

References

[1] 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*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022. URL: https://openreview.net/pdf?id=8XWP2ewX-im.

[2] X. Davies, L. Langosco, and D. Krueger, *Unifying grokking and double descent*, 2022. URL: https://openreview.net/pdf?id=JqtHMZtqWm.

[3] Z. Liu, E. J. Michaud, and M. Tegmark, *Omnigrok: Grokking beyond algorithmic data*, 2023. URL: https://openreview.net/pdf?id=zDiHoIWa0q1.

[4] W. Merrill, N. Tsilivis, and A. Shukla, *A tale of two circuits: Grokking as competition of sparse and dense subnetworks*, 2023. URL: https://openreview.net/pdf?id=8GZxtu46Kx.

[5] A. Power, Y. Burda, H. Edwards, I. Babuschkin, and V. Misra, *Grokking: Generalization beyond overfitting on small algorithmic datasets*, 2022. arXiv: 2201.02177 [cs.LG]. URL: https://arxiv.org/pdf/2201.02177.pdf.

[6] 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*, 2022. arXiv: 2206.04817 [cs.LG]. URL: https://arxiv.org/pdf/2206.04817.pdf.

[7] V. Varma, R. Shah, Z. Kenton, J. Kramár, and R. Kumar, *Explaining grokking through circuit efficiency*, 2023. arXiv: 2309.02390 [cs.LG]. URL: https://arxiv.org/pdf/2309.02390.pdf.