Grokking in neural networks and the role of regularization

By

Ahmed Mohammed Ahmed, (ahmedm@aims.edu.gh)
Supervised by: Dr.Tiffany Vlaar

June 2024

AN ESSAY PRESENTED TO AIMS-GHANA IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTER OF SCIENCE IN MATHEMATICS

DECLARATION

This work was carried out at AIMS-Ghana in partial fulfilment of the requirements for a Master of Science Degree.

I hereby declare that except where due acknowledgement is made, this work has never been presented wholly or in part for the award of a degree at AIMS-Ghana or any other University.

Student: AHMED RASHID ABDELHAID MOHAMMED AHMED

Supervisor: Dr.Tiffany Vlaar

ACKNOWLEDGEMENTS

First and foremost, I thank Almighty Allah for his blessings showered on me in completing this research successfully. I would like to express my deep gratitude to my supervisor, Dr. Tiffany Vlaar, who provided me with the invaluable opportunity to undertake this remarkable research. Her encouragement, suggestions, mentorship, and guidance have been immensely appreciated. I am also deeply thankful to the AIMS Ghana community with its wonderful team management and academic staff who provided me with the outstanding study environment that facilitated the production of this research. My final appreciation goes to all my friends, for their understanding and encouragement.

Abstract

Typically, as a neural network's performance on training data improves, its performance on (unseen) validation data also improves until the network starts overfitting.However, grokking, a recently discovered phenomenon by [26], defies this norm by showing a sudden and dramatic improvement in validation performance long after overfitting has begun. This delayed generalization pattern has been notably observed in algorithmic tasks like modular addition. Grokking is sometimes viewed as a phase change in the network's learning process. In this project, we explore the concept of grokking in the context of modular addition, examining its occurrence in different types of neural networks and training setups. We discovered that groking happens in many types of models and is influenced by the amount of training data used. Specifically, we conduct our experiments in recurrent neural networks (RNNs) and long short-term memory (LSTM) networks. Additionally, our results show that using regularization techniques such as weight decay can help the models generalize better. Our observations indicate that training without weight decay leads to poor generalization and that removing weight decay too early also has the same effect. Moreover, we demonstrate that weight decay is not necessary during the early phases of training. Introducing weight decay only late in training still leads to grokking, which is a novel discovery. These experiments aim to provide deeper insights into the factors that affect grokking.

Contents

Declaration i

Acknowledgements ii

Abstract iii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Objective . . . 2

1.3 Research Questions . . . 2

2 Machine learning Background 3

2.1 Mathematical Model of a Neuron . . . 3

2.2 Neural Network Training . . . 6

2.3 Neural Networks Architecture . . . 12

3 Overfitting Problem 18

3.1 Regularization and Overfitting . . . 19

4 Reality of Deep learning 21

4.1 Double Descent . . . 21

4.2 Grokking . . . 23

4.3 Grokking Literature Review . . . 24

5 Methodology and Experimental Results 26

5.1 Role of architecture and Training Set Size . . . 28

5.2 Experiment: The Role of Regularization . . . 30

5.3 Summary of Results . . . 33

5.4 Finding 1: Impact of Sample Size on Grokking . . . 34

5.5 Finding 2: Impact of Weight Decay on Grokking . . . 34

6 Conclusion and Further Work 35

References 38

APPENDICES 39

7 appendices 39

7.1 Notations and Setup 39

7.2 Weight Decay 45

7.3 Weight Initialization Method 48

1. Introduction

No one can deny the critical role that machine learning plays in our modern life. Its impact spans various domains, from healthcare and finance to entertainment and communication, changing how we interact with technology and the world around us. Power et al. [26] recent discovery of the Grokking phenomenon in 2022 is a quite interesting observation in this field. It describes the sudden improvement in a model's generalization performance after extended training on synthetic datasets. Initially, the model overfits the training dataset, showing high training accuracy but low validation accuracy on unseen data; however, after continued training with significantly more optimization steps,the model suddenly begins to generalize well to the validation set, achieving near-perfect validation accuracy [26].

1.1 Motivation

One of the primary motivations for studying grokking is its potential to enhance the interpretability of machine learning models. Many deep neural networks are considered " black boxes " because of their complexity and lack of transparency [23]. This makes it difficult to understand and explain the behavior of these models, which is essential for ensuring their reliability and safety.

The grokking phenomenon, along with other interesting behaviors observed in machine learning such as the double descent phenomenon[22], presents a challenge to classical machine learning theories. Both grokking and double descent defy traditional machine learning theories since classical theories suggest that once a model overfits the training data, it will perform poorly on new, unseen data [7]. However, double descent demonstrates that increasing the model's capacity beyond a certain point can lead to improved performance on validation data, contrary to the expectations set by traditional bias-variance tradeoff concepts shown in 4.1. Similarly, grokking reveals that extended training in an over-parametrized setting can result in sudden and significant improvements in generalization performance. Grokking and double decent phenomenas indicate the need to update our understanding of model capacity, overfitting, and generalization. By studying grokking, we can gain understanding how models generalize which will improve our approaches in designing and training machine learning systems. Moreover, the emergent behavior observed in large language models (LLMs)[11] is similar to grokking to some extent . As LLMs grow in size and complexity, they show new and unexpected abilities, similar to the sudden improvement seen in grokking. Investigating grokking may help us understand how LLM models gain those abilities, offering new ways to enhance model performance and generalization in a controlled and predictable manner.

1.2 Objective

Investigate the grokking phenomenon across various models for example (Transformer [35], Recurrent Neural Networks [29] and Long Short Term Memory [10]) and analyze the impact of regularization and samplesize on grokking.

1.3 Research Questions

- Do different types of neural networks architecture exhibit grokking behavior ?
- Studying the effect of training data set size for different architectures ?
- How do turning regularization off effect grokking phenomenon ?

2. Machine learning Background

This chapter will provide an overview of key concepts and techniques in neural networks. The discussion will be organized into the following topics: Mathematical Model of a Neuron, A Multilayer Perceptron (MLP) , Backpropagation in Neural Networks, Neural Networks Architecture: Recurrent Neural Networks (RNN), Long Short Term Memory (LSTM) and Transformer Architectures.

2.1 Mathematical Model of a Neuron

An artificial neural network draws inspiration from real human thinking, which is capable of performing non-trivial tasks that classical programs cannot do. Therefore, it is reasonable to turn to neurobiology and draw ideas from real human brain neurons [38]. Figure 2.1 reflects this analogy in more detail.

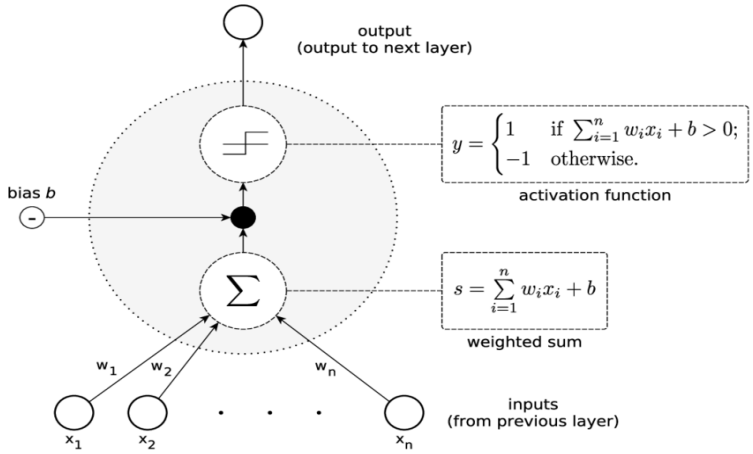


Figure 2.1: This Figure illustrates the components of a single perceptron in a neural network. The perceptron receives inputs $(x_1, x_2, \ldots, x_n)$ from the previous layer, each associated with a weight $(w_1, w_2, \ldots, w_n)$. These inputs are combined into a weighted sum $(s = \sum_{i=1}^{n} w_i x_i + b)$, where b is the bias term. This sum is then passed through an activation function, which in this case is a step function, to produce the final output (y). The activation function outputs 1 if the weighted sum exceeds zero and -1 otherwise, allowing the perceptron to make binary classifications [33].

Input data is received by a neuron, which can be considered as a "black box" that produces a specific result y. The mathematical neuron has a summing unit where each input signal is multiplied by a certain real weight coefficient, and the final sum is formed. The obtained value is passed to the activation function, which determines whether the neuron is activated or not. If we have a large number at the input, passing it to the activation function will result in a number in the desired range (usually $[0, 1]$ or $[-1, 1]$) [38].

2.1.1 Activation Functions.

Activation functions are important in neural networks as they introduce nonlinearity into the model. Common activation functions include the Rectified Linear Unit (ReLU 2.1.3), sigmoid (2.1.1) , and tanh functions (2.1.2) [37]. These functions are applied element-wise to the outputs of the hidden layers, transforming the linear combinations into nonlinear representations. This nonlinearity allows the network to capture and model complex patterns within the data, far beyond what linear models can achieve. In designing a neural network, one of the key decisions you make is which activation functions to use for the individual layers and the output unit.

Sigmoid Function is an activation function represented as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.1.1}$$

It maps input values between 0 and 1.

Hyperbolic Tangent (tanh) Function. A commonly better alternative for hidden layers is the hyperbolic tangent (tanh) function:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.1.2}$$

This function maps input values between -1 and 1, centering the activations around zero, which often leads to faster and more efficient learning compared to the sigmoid function [37].

Rectified Linear Unit (ReLU).

$$\text{ReLU}(x) = \max(0, x). \tag{2.1.3}$$

ReLU is preferred for hidden layers because it allows for quicker learning. However, its downside is that neurons can sometimes "die" if they get stuck with negative input values [37].

Softmax Function transforms the outputs into probabilities that sum to 1, making them interpretable as a probability distribution over classes [37].

$$\text{softmax}(a_i) = \frac{e^{a_i}}{\sum_j e^{a_j}}. \tag{2.1.4}$$

where a represents the raw output values (logits) from the neurons in the network before applying the softmax transformation.

Neural Networks is a collection of nodes or "neurons" that are connected in a structured way. Each neuron holds a number, usually between 0 and 1, called an activation we will disuss more about it in the following section 2.1.2.

2.1.2 A Multilayer Perceptron (MLP).

A multilayer perceptron (MLP) consists of multiple layers of neurons, each fully connected to the next one. The basic structure includes an input layer, one or more hidden layers, and an output layer. The input layer accepts raw data, while the hidden layers process the data through a series of transformations, culminating in the output layer, which generates the final predictions [38].

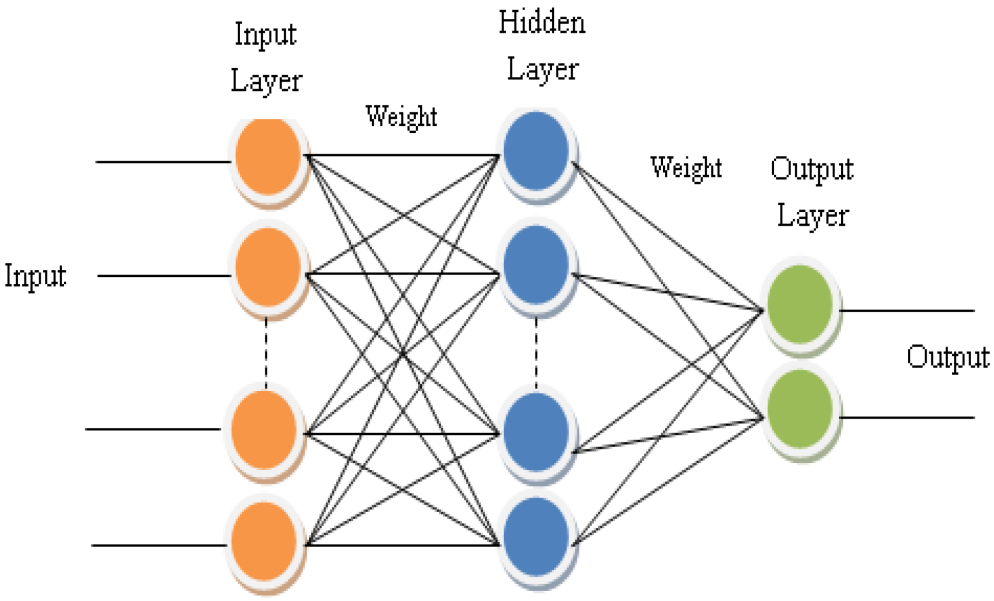


Figure 2.2: Multilayer Perceptron [31].

The connections between neurons in different layers are controlled by mathematical operations:

- **Weighted Sum:** Each neuron computes a weighted sum of the activations from the previous layer's neurons. Each activation is multiplied by its corresponding weight and summed up [37].
- **Activation Function:** This sum is passed through an activation function, such as the sigmoid 2.1.1 function, which converts the sum into a value between 0 and 1. This determines the neuron's activation in the next layer [37].

To further enhance the model's expressive power, we can stack multiple hidden layers. Each additional layer applies further transformations, enabling the network to learn increasingly abstract representations of the input data [38].

2.2 Neural Network Training

Training a neural network is the process of searching for model parameters that transform input data into the desired output information. However, the objective of a neural network is not simply to provide the perfectly correct answer through weight adjustment. The expectation is to achieve much greater results - an algorithm capable of generalizing to unseen data. Two sets are distinguished: the training set (on which the algorithm is trained) and the test set (on which the effectiveness of the algorithm is evaluated) [38].

Learning Process

An epoch means the model has seen all examples from the training set once [38]. During training, the learning algorithm adjusts the weights to minimize the loss function. The primary task of the learning algorithm is to adjust the model parameters so that the loss function tends to its minimum [38].

Hyperparameters

Hyperparameters are adjustable parameters that control the model training process [38].

Error and Loss Function

Error is a quantity that reflects the discrepancy between the expected and obtained answers [38]. The error computed should decrease over time. If this does not happen, it is necessary to either change the model architecture or modify the hyperparameters. There are many ways to estimate the error, such as calculating the loss function, for example, the Mean Squared Error (MSE).

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{2.2.1}$$

- n: The number of observations or data points.
- y_i: The true value (actual value) of the i-th observation.
- $\hat{y}_i$: The predicted value for the i-th observation provided by the model.
- $(y_i - \hat{y}_i)$: The error or residual for the i-th observation, which is the difference between the true value and the predicted value.
- $(y_i - \hat{y}_i)^2$: The squared error for the i-th observation.

2.2.1 Backpropagation in Neural Networks.

The backpropagation algorithm computes the gradient of the loss function with respect to each weight in the network by iteratively applying the chain rule of calculus [28].

for $i = 1$ *to epochs* **do**
  // Number of complete passes through the training dataset **for** *each training example* $(\mathbf{x}, y)$ **do**
    // Input feature vector $\mathbf{x}$ and target output y Forward propagation **for** *each neuron* i *in input layer* **do**
      $a_i^{(0)} \leftarrow x_i$ // Activation of neuron i in input layer set to input feature x_i
    for $l = 1$ *to* L **do**
      // Loop through each layer l **for** *each neuron* i *in layer* l **do**
        Compute $z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)}$ // Weighted sum of inputs to neuron i in layer l
        Compute $a_i^{(l)} = f(z_i^{(l)})$ // Activation of neuron i in layer l
    Backward propagation **for** *each neuron* i *in output layer* **do**
      Compute $\delta_i^{(L)} = \frac{\partial \mathcal{L}(y_i, o_i)}{\partial o_i} f'(z_i^{(L)})$ // Error term for neuron i in output layer
    for $l = L - 1$ *to 1* **do**
      // Loop backward through each layer **for** *each neuron* i *in layer* l **do**
        Compute $\delta_i^{(l)} = f'(z_i^{(l)}) \sum_j w_{ji}^{(l+1)} \delta_j^{(l+1)}$ // Error term for neuron i in layer l
    Update weights **for** *each weight* $w_{ij}^{(l)}$ **do**
      Update weight $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \alpha \delta_i^{(l)} a_j^{(l-1)}$
      // Adjust weight by learning rate α, error term $\delta_i^{(l)}$, and activation $a_j^{(l-1)}$

1. Initialization: We initialize the weights and biases of the neural network randomly.
2. Forward Pass: We feed the input data through the network, layer by layer, to produce output by applying weights and activation functions at each layer to transform it into output [30].
3. Calculate Loss: We compare the output generated by the network to the actual target output. The difference between these two values is measured using a loss function, quantifying how well the network performs [30].
4. Backward Pass: We propagate the error (the difference between the actual and predicted output) backward through the network to determine how much each weight contributed to the error. We update the weights using the gradients calculated in the backward pass. We use an optimization algorithm like gradient descent, which adjusts the weights in the direction that reduces the loss [30].
5. Iterate: We repeat the process of forward pass, calculating loss, backward pass, and adjusting weights many times. With each iteration, the network's predictions should get closer to the output targets, which ultimately improves accuracy [30].

2.2.2 Learning Algorithms.

The essence of ordinary gradient descent is to minimize the loss function by taking steps in the direction of the steepest descent of the function. For this, the fact that the vector of partial derivatives of the function $f(x) = f(x_1, \ldots, x_n)$ sets the direction of the steepest ascent of this function is used [4]. In this case, moving along the antigradient sets the fastest decrease of the function. The problem with this approach is that summing certain values for each object requires a lot of time in practice for large tasks, the method needs to be modified. We only use only one element , but not the entire dataset [4].

$$w^{(t+1)} = w^{(t)} - \alpha \frac{1}{l} \sum_{i=1}^{l} \nabla L_i(w^{(t)}). \quad (\text{ Gradient Decent}) \tag{2.2.2}$$

$$w^{(t+1)} = w^{(t)} - \alpha \nabla L_i(w^{(t)}). \quad (\text{Stochastic Gradient Decent}) \tag{2.2.3}$$

where w, α are the weight coefficient and the learning rate respectively, determining how strongly the weight vector changes when shifting along the gradient and l is the total number of training example [4].

Mini-Batch Learning.

Mini-batch learning is a variant of stochastic gradient descent (SGD). Instead of using the entire training set or a single example, mini-batch learning processes small subsets of the training data, known as mini-batches. This approach balance between the high variance of single-example updates and the computational expense of full-batch updates. Each mini-batch can be viewed as a sample drawn from the training set, which itself is a sample from the overall population [27].

$$w^{(t+1)} = w^{(t)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla L_i(w^{(t)}). \quad (\text{Mini-Batch Gradient Descent}) \tag{2.2.4}$$

During mini-batch learning, the forward and backward passes are computed on each mini-batch. This introduces some noise into the gradient estimation, as the gradients are computed on smaller, potentially less representative samples. While this noise can lead to less stable convergence, it also has beneficial properties. Specifically, it helps in escaping local minima and exploring the loss landscape more thoroughly, which is particularly valuable in the context of non-convex loss functions common in deep learning [27].

The balance of noise introduced by mini-batches is both a strength and a challenge. The noisier gradients can prevent the algorithm from getting stuck in local minima, but they can also cause oscillations and slow convergence [27]. Techniques such as learning rate decay can mitigate these oscillations which we will discuss in the following section .

Learning Rate Decay.
Learning rate decay involves gradually reducing the learning rate over the course of training. This technique aims to find a balance between large initial learning rates, which enable faster learning and exploration, and smaller learning rates later on, which allow for fine-tuning our model and convergence [36].

A large learning rate at the beginning helps the model make significant progress towards the global minimum by taking larger steps. However, if maintained throughout the training process, it can lead to overshooting and instability, preventing the model from converging . On the other hand, a small learning rate throughout the training can cause the model to take very small steps, potentially getting stuck in local minima and leading to inefficient learning [36].

In deep learning, optimization algorithms face challenges due to the landscape of the loss function . Convex optimization deals with problems where the objective function and possible solutions form a smooth, predictable shape, making them easier to solve. However, non-convex optimization, which is common in deep learning, can have many local minima and maxima, meaning that finding a low point does not necessarily indicate that it is the lowest point overall as shown in Figure 2.3. In high-dimensional spaces, like those in deep learning most points where the gradient (slope) is zero are not local optima but saddle points. A saddle point, dips in the middle but rises at the front and back.

In high-dimensional spaces, these saddle points are much more common than local minima [43].

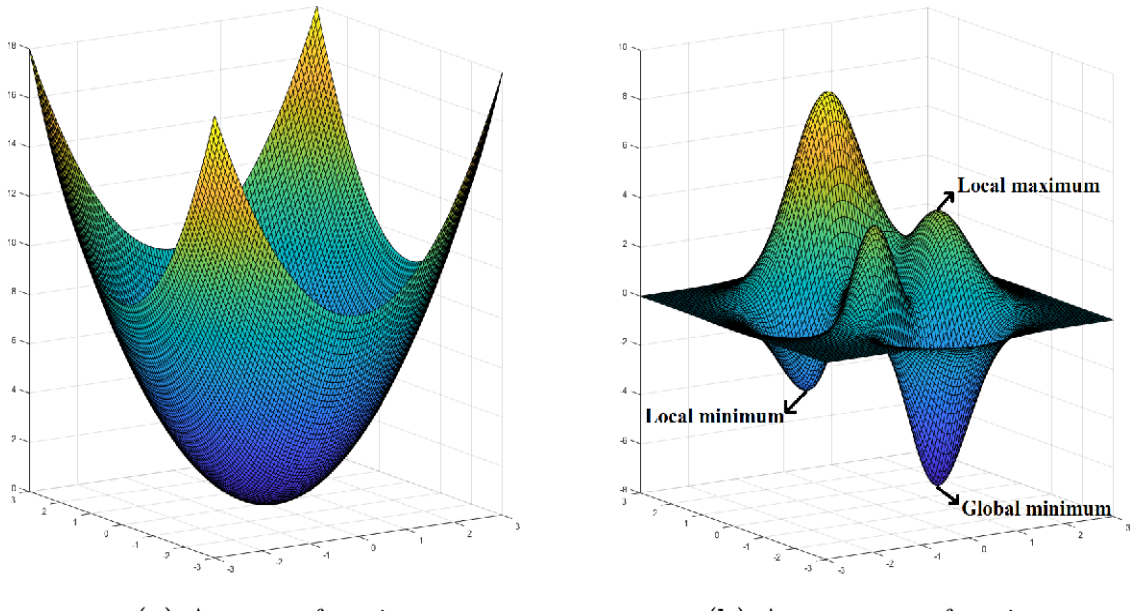


Figure 2.3: convex (left) and non-convex (right) functions. Convex functions have a single global minimum and no saddle points. In contrast, non-convex functions, have many saddle points and local minima, making optimization more challenging [43].

Advanced optimization techniques such as Adam and Gradient Descent with Momentum have been developed to address these challenges, helping the optimization process move faster through flat regions and achieve better convergence which we will discuss in the following section 2.2.3.

2.2.3 Adam (Adaptive Moment Estimation).

Adam stands for Adaptive Moment Estimation. It combines the exponentially moving average of the first and the second moment to make updates. This combination allows Adam to adapt the learning rates for each parameter, resulting in a more efficient and stable optimization process [14].

Algorithm 1: Adam [14].

Input: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$
Data: initialize $t \leftarrow 0$, $\theta_0 \in \mathbb{R}^n$, $m_0 \leftarrow 0$, $v_0 \leftarrow 0$
repeat
  $t \leftarrow t + 1$;
  $\nabla f_t(\theta_{t-1}) \leftarrow$ SelectBatch(θ_{t-1}) // select batch and return gradient
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1)\nabla f_t(\theta_{t-1})$ // element-wise
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)(\nabla f_t(\theta_{t-1}))^2$;
  $\hat{m}_t \leftarrow m_t/(1 - \beta_1^t)$;
  $\hat{v}_t \leftarrow v_t/(1 - \beta_2^t)$;
  $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$;
until *stopping criterion is met*;
return optimized parameters θ_t;

The main idea behind Adam is to use estimates of the first and second moments of the gradients to adapt the learning rate for each parameter. The parameters used in the algorithm are:

- α: Learning rate for Adam.
- β_1: Exponential decay rate for the first moment estimates.
- β_2: Exponential decay rate for the second moment estimates.
- ϵ: A small constant for numerical stability to avoid division by zero .

Gradient Descent with Momentum.

The main idea of gradient descent with momentum is to compute a weighted average of past gradients, which helps smooth out the optimization steps. By adding momentum, the algorithm maintains a more consistent direction of movement, reducing oscillations and allowing for faster progress towards the minimum of the cost function [14].

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t. \tag{2.2.5}$$

Here, m_t is the exponentially weighted average of past gradients (momentum) at iteration t, and g_t is the gradient of the cost function at iteration t.

Inspiration from RMSprop.
Adam draws inspiration from RMSprop, which stores an exponentially moving average of squared gradients to adjust the learning rates for each parameter based on the history of their gradients [14].

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \tag{2.2.6}$$

Here, v_t is the exponentially weighted average of squared gradients at iteration t, and β_2 is the exponential decay rate for the second moment estimates.

During updates, the gradient is divided by the square root of this average, effectively dampening oscillations in directions with large gradients and allowing for faster convergence.

Problems with Weight Decay in Adam

AdamW uses Weight Decay regularization, which adds a penalty to the loss function proportional to the square of the magnitude of the model weights. This helps prevent overfitting by discouraging large weights, leading to simpler models that generalize better to unseen data. We will discuss this in more detail in section 3.1.

When Adam includes weight decay in the gradient updates, it can sometimes cause problems:

The weight decay can interfere with the adaptive learning rates, making the training less efficient. **Overshooting:** The model might take steps that are too large, leading to instability and poorer performance [14].

AdamW (Adam with Weight Decay) makes a small but important change:

Decoupling: It separates the weight decay from the gradient updates. Instead of combining weight decay with the gradient updates, AdamW applies weight decay directly to the weights themselves after the gradient step [18].

AdamW Update Rule:

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \left(\frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) - \eta_t \lambda \theta_{t-1}. \tag{2.2.7}$$

This equation represents the AdamW update, where:

- α is learning rate .
- The term $\left(\frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right)$ is the same as in Adam and handles the adaptive gradient update.
- The term $-\eta_t \lambda \theta_{t-1}$ is the weight decay (regularization) term applied directly to the parameters.

2.3 Neural Networks Architecture

2.3.1 Recurrent Neural Networks (RNN). Traditional neural networks (feed-forward networks) as shown in 2.1.2 have an input layer, hidden layers, and an output layer. RNNs enhance this by adding loops, enabling them to pass information forward and maintain a hidden state representing previous inputs [29]. In an RNN, each unit (or cell) processes an element of the input sequence along with the hidden state from the previous step. The hidden state acts as the network's memory, capturing information about the sequence up to that point [29]. At the core of RNNs is the idea of maintaining a hidden state that is updated at each time step. The hidden state captures information from previous time steps, allowing the network to retain memory of past inputs [29]. The equation that defines the update of the hidden state in a simple RNN is:

$$H_t = \phi_h(X_t W_{xh} + H_{t-1} W_{hh} + b_h). \tag{2.3.1}$$

Since H_t recursively includes H_{t-1} and this process occurs for every time step, the RNN includes traces of all hidden states that preceded H_{t-1} as well as H_{t-1} itself [29]. The equation for the output variable in a simple RNN is:

$$O_t = \phi_o(H_t W_{ho} + b_o). \tag{2.3.2}$$

where:

- H_t: The hidden state at time step t. X_t :The input state at time step t.
- H_t: The hidden state at time step t.
- ϕ_h: The activation function (e.g., tanh 2.1.2 or ReLU 2.1.3) applied element-wise.
- ϕ_0: The activation function applied to the output gate (e.g., Sigmoid 2.1.1) applied element-wise.
- W_{xh}: The weight matrix for the input X.
- W_{hh}: The weight matrix for the previous hidden state H_{t-1}.
- W_{ho}: The weight matrix for the hidden state H.
- b_h: The bias term for the hidden state.
- b_o: The bias term for the output.

The hidden state H_t is a function of the current input X_t and the previous hidden state H_{t-1}. This recursive nature enables the RNN to maintain a dynamic memory of past information [29].

Challenges with RNNs. One significant challenge with vanilla RNNs is their difficulty in learning long-term dependencies due to the vanishing and exploding gradient problem. As gradients are backpropagated as shown in 2.2.1 through many time steps, they tend to become very small, effectively preventing the network from learning long-range patterns [10].

To address these challenges, advanced architectures like Long Short-Term Memory (LSTM) is developed. The LSTM introduces gating mechanisms that regulate the flow of information, making it easier to capture long-term dependencies [10]. We will discuss more about LSTM in the next section 2.3.2.

2.3.2 Long Short-Term Memory (LSTM).

Long Short-Term Memory (LSTM) units consist of a cell state and three gates: input, forget, and output. These gates control the flow of information into and out of the cell state, helping the network retain relevant information over long periods and forget irrelevant information. The cell state acts as a memory that carries information across many time steps. The gates, which use sigmoid activations (σ), decide how much of the new input, the previous cell state, and the previous hidden state should be used to update the current cell state and the hidden state [10]. The three key gates are the forget gate, input gate, and output gate [10], as shown in Figure 2.4.

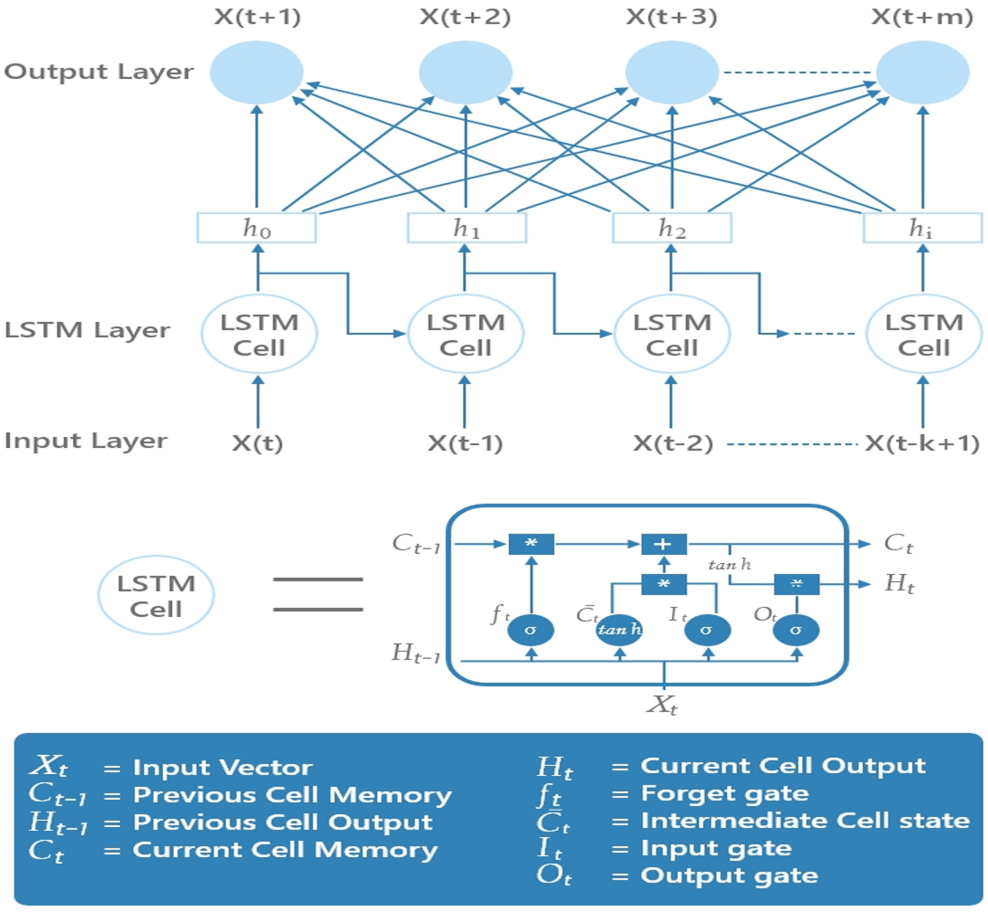


Figure 2.4: LSTM Model Architecture [5].

Input Processing: At each moment in time t, the LSTM receives an input X_t and the previous hidden state H_{t-1} [29].

Forget Gate decides what to keep or forget from the previous memory. It looks at the previous hidden state H_{t-1} and the current input X_t to decide [29]. It uses a sigmoid function to produce a value between 0 and 1. If it's close to 0, it means forget, and if it's close to 1, it means keep . It computes the forget gate vector F_t:

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f). \tag{2.3.3}$$

Input Gate decides what new information to store in the memory cell. It checks both the previous hidden state H_{t-1} and the current input X_t. It uses two functions: a sigmoid function to decide which values to update and a hyperbolic tangent function to generate new candidate values [29] . It computes the input gate vector I_t and the candidate values $\tilde{C}_t$:

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i). \tag{2.3.4}$$

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c). \tag{2.3.5}$$

Update Cell State: This operation merges the old memory with the new candidate values to create the updated memory. It decides how much of the old memory to keep using the forget gate and how much of the new candidate values to add using the input gate. This is done through element-wise multiplication ($\odot$) and addition [29].

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t. \tag{2.3.6}$$

Output Gate: The output gate controls what information to output from the memory cell. It regulates the flow of information from the memory to the next hidden state. It looks at both the previous hidden state H_{t-1} and the current input X_t to decide what to output. It uses a sigmoid function to make decisions. The output gate vector O_t is computed as:

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o). \tag{2.3.7}$$

Next Hidden State: Finally, the LSTM combines the updated memory C_t with the output gate O_t to produce the next hidden state H_t. This is done using another hyperbolic tangent function.

$$H_t = O_t \odot \tanh(C_t). \tag{2.3.8}$$

Recurrent Connections: The process repeats for each time step, with the hidden state H_{t-1} becoming the input for the next time step [29].

Final Output: After processing all time steps, the LSTM may produce an output at each time step or only at the final time step, depending on the task. The output can be used for various applications such as sequence prediction, classification, or generation [29].

2.3.3 Transformer Model. Transformers utilize the attention mechanism, which enables them to process entire sequences of data simultaneously, unlike RNNs which process data sequentially. This gives transformers a significant advantage, especially for tasks involving long sequences where RNNs face limitations due to their short-term memory. While LSTMs have improved memory capabilities, they still cannot match the performance of transformers [35].

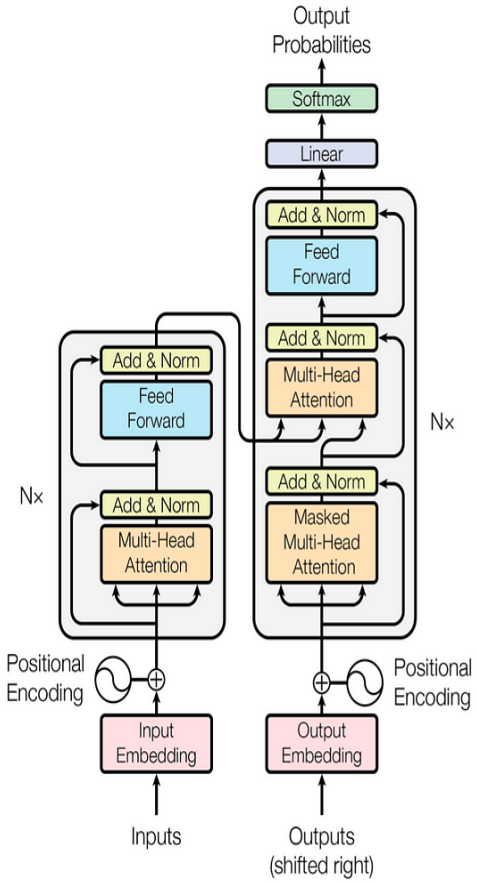


Figure 2.5: Trasnformer Model Architecture [35].

Key Concepts from Transformer architecture [35]:

Goal of the Model:

The model's primary task is to predict the next word in a sequence given some input text. The input text is divided into tokens, which for simplicity, we will consider as whole words.

1. **Token Embedding**: Each token t in the input sequence is converted into a high-dimensional vector called an embedding e:

$$e = \text{Embedding}(t).$$

This process maps the token to a space where similar meanings are close together.

2. **Initial Embeddings**: These embeddings are the starting point and represent the individual meanings of the tokens without considering the context.

3. **Queries, Keys, and Values**: For each token embedding, we compute three vectors: Query (Q), Key (K), and Value (V) using weight matrices W_Q, W_K, and W_V:

$$\mathbf{Q} = W_Q\mathbf{e}, \quad \mathbf{K} = W_K\mathbf{e}, \quad \mathbf{V} = W_V.\mathbf{e}. \tag{2.3.9}$$

4. **Dot Products and Relevance Scores**: We calculate the dot product of the Query and Key vectors to determine the relevance score:

$$\text{score}(q_i, k_j) = \mathbf{q}_i \cdot \mathbf{k}_j. \tag{2.3.10}$$

5. **Softmax and Attention Weights**: Apply the softmax function (2.1.1) to the relevance scores to convert them into probabilities (weights) that sum to 1. This forms the attention pattern:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2.3.11}$$

6. **Value Vectors and Updates**: The Value vectors are multiplied by the attention weights. Each value vector is scaled and summed to produce a new embedding that incorporates contextual information.

7. **Attention Pattern**: This pattern shows how much attention each word pays to every other word.

8. **Weighted Sum of Values**: The new embedding for each token is a weighted sum of the value vectors, where weights are the attention scores:

$$\text{New\_embedding}_i = \sum_j \text{Attention}(q_i, k_j). \tag{2.3.12}$$

9. **Masked Attention**: During training, a mask is applied to prevent future tokens from influencing the prediction of earlier tokens:

$$\text{Mask}(i, j) = \begin{cases} 0 & \text{if } i \leq j. \\ -\infty & \text{if } i > j. \end{cases} \tag{2.3.13}$$

This ensures that the model only attends to previous or current tokens.

10. **Multi-Headed Attention**: Multiple sets of Q, K, and V matrices (heads) are used in parallel to capture different types of relationships. The outputs of all heads are combined to form the final embeddings.

2.3.4 Batch Normalization.

Batch normalization is a technique to improve the training of deep neural networks by normalizing the inputs of each layer. It helps stabilize and accelerate the learning process by maintaining the mean and variance of the inputs within a layer at a constant level. When we train a model, we normalize the input features to have a mean of 0 and a variance of 1. This normalization helps the training algorithm to converge faster. Similarly, in deep neural networks, normalizing the inputs of each layer can make the training more efficient [12].

Batch normalization normalizes the outputs of the previous layer before passing them to the next layer. It works by first calculating the mean and variance of the inputs for each mini-batch. Then, it uses these statistics to normalize the inputs. The process involves the following steps:

1. We compute the mean of the batch μ_B:

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i. \tag{2.3.14}$$

where x_i represents the input values of the layer, and m is the number of inputs in the mini-batch.

2. We compute the variance of the batch σ_B^2:

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2. \tag{2.3.15}$$

3. We normalize the inputs:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}. \tag{2.3.16}$$

here, ϵ is a small constant added to avoid division by zero.

4. We scale and shift the normalized inputs:

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i). \tag{2.3.17}$$

where γ and β are learnable parameters that allow the network to scale and shift the normalized inputs to better fit the data.

γ and β ensure that the network can still represent complex functions by adjusting the normalized values appropriately. They are important because they allow the model to adjust the normalized hidden unit values to the distribution that best suits the learning process. This helps in effectively utilizing the non-linearity of activation functions such as the sigmoid function, which performs better when values are spread out rather than clustered around a particular value [12].

3. Overfitting Problem

In this chapter, we will discuss the critical issue of overfitting in machine learning models and its impact on model performance, and the methods employed to mitigate it.The Reality of deep learning will be discuss in next Chapter 4.

Model tuning consists of selecting the weight (coefficients of the features) and and biass in such a way that the loss function takes its minimum value on both the training and test sets.The problem of overfitting arises when a model, in its attempt to minimize training error, loses its ability to generalize to new, unseen data [41]. In the context of a classification task as shown in Figure 3.1.

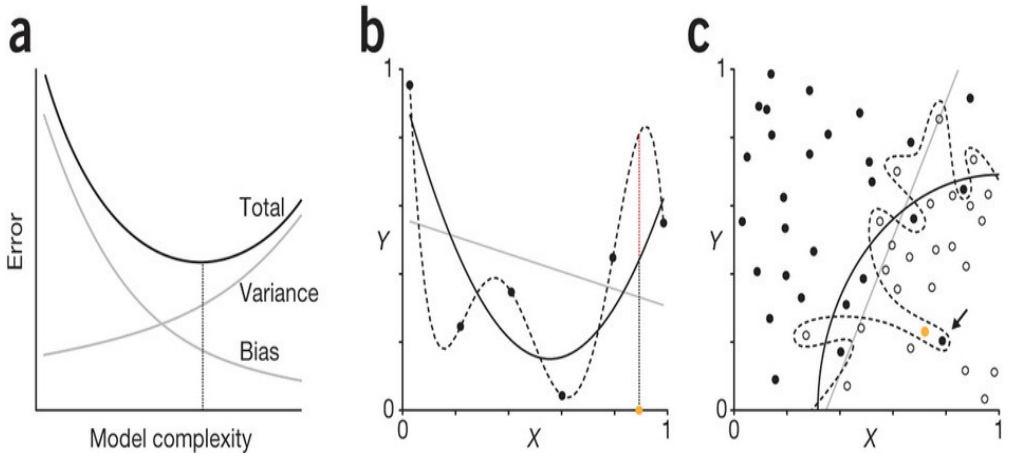


Figure 3.1: Illustrations of the bias-variance tradeoff, underfitting vs. overfitting, and the impact of model complexity on generalization [15].
(a) Bias-Variance Tradeoff:Reducing bias increases variance and vice versa. The goal is to find a balance with low bias and low variance for optimal performance.
(b) This graph contrasts underfitting (high bias), optimal fitting (balance between bias and variance), and overfitting (high variance).
(c) **Underfitting**: The straight gray line represents a model that approximately separates the objects into two classes but misclassifies some elements, indicating underfitting. This model has high bias and fails to capture the underlying pattern in the data.
Optimal Model Fit: The black curve represents an optimally fitted model that captures the overall trend of the data. This model generalizes well to unseen data (open circles) by learning the underlying patterns without overfitting to the training data.Yellow Point: Indicates the point of optimal model complexity where the total error is minimized. At this point, the balance between bias and variance is ideal, resulting in the lowest possible error.
Overfitting: The dashed black curve represents an overfitted model that closely follows the training data points (black dots), including the noise, achieving zero training error. However, this perfect separation is only observed on the training dataset. When evaluated on the test set, the overfitted model performs poorly because it has memorized the training data rather than learning the underlying patterns that generalize to new data the orange point.

3.1 Regularization and Overfitting

Regularization is a technique used to control overfitting in machine learning models, including deep neural networks .Regularization helps by penalizing the model complexity, encouraging simpler models that generalize better [1].

Cost Function with Regularization. ℓ_2 regularization, also called weight decay, is one of the most commonly used regularization techniques. It adds an extra term to the cost function that penalizes large weights. We will consider a neural network with a cost function $\mathcal{L}(\mathbf{w})$, which is the sum of the losses over the training data.

$$\mathcal{L}_\lambda(\mathbf{w}) := \mathcal{L}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{3.1.1}$$

where λ is the regularization parameter, and $\|\mathbf{w}\|^2$ is the L2 norm . This term discourages the weights from becoming too large [1].

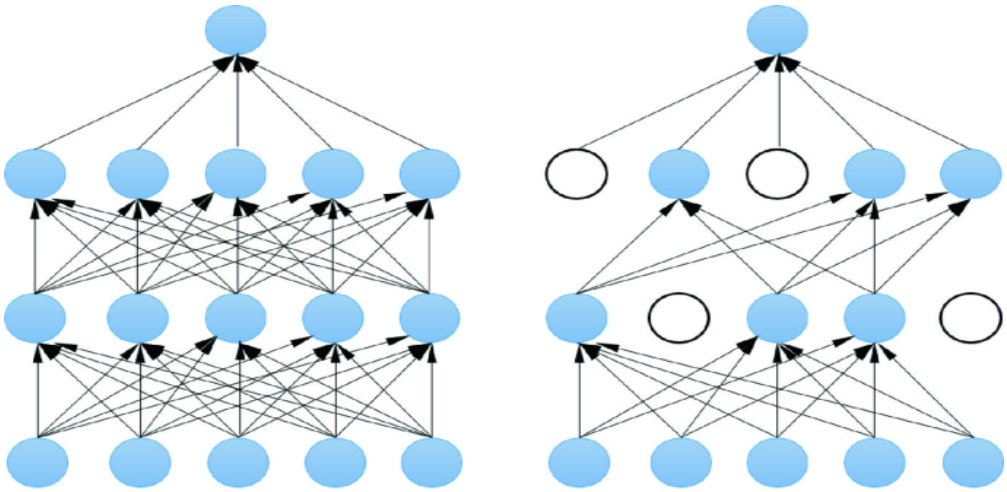
Why Regularization Helps with Overfitting ?

1. **Shrinkage of Weights**: By increasing the regularization parameter λ, the weight vector $\mathbf{w}$ is incentivized to be smaller. When the weights are close to zero, the model becomes simpler. This means it starts ignoring less important features, which reduces the model's capacity to overfit the training data [1].
2. **Linear Approximation**: With strong regularization, the deep neural network approximates a linear function. Even if the network is deep, its capacity to model highly complex, non-linear relationships is reduced . This simplification reduces overfitting, as the network cannot fit the noise in the training data [1].

Dropout is another regularization technique that helps control overfitting by randomly setting a fraction of the input units to zero at each update during training time. Which will be discussed more in the following section 3.1.1.

3.1.1 Dropout.

Dropout is a regularization technique in neural networks that randomly disables units during training, simulating the effect of training multiple smaller networks as shown in Figure 3.2 [32]. Dropout acts as a form of regularization by effectively training the network with smaller sub-networks on each iteration. This process helps prevent overfitting by ensuring that the network doesn't become overly reliant on specific units or features. From the perspective of individual units within the network, dropout encourages them to distribute their weights more evenly across the inputs. Since any input or influence could be randomly eliminated during training, units become less inclined to rely heavily on any single feature. This encourages a more generalized representation of the data and helps prevent the network from memorizing noise in the training data [32].



(a) A standard deep neural network. (b) After applying dropout.

Figure 3.2: Dropout [32].

In terms of implementation, adjusting the dropout rate (keep probability) for different layers can further enhance its effectiveness. Layers where overfitting is a concern can have lower dropout rates, while layers with less risk of overfitting can have higher dropout rates or even none at all. This adaptive dropout strategy is akin to tuning the regularization parameter in L2 regularization to different degrees for different layers[32]. However, dropout does come with its challenges. One notable downside is that it makes the cost function less well-defined on every iteration, which can complicate the monitoring of training progress. Debugging with dropout can be trickier since the cost function may not consistently decrease on each iteration [32].

4. Reality of Deep learning

Traditionally, increasing model complexity beyond a certain point leads to overfitting, where the model performs well on training data but poorly on unseen data . However, double descent and grokking show that after this initial overfitting phase, further increasing the model complexity can lead to improved generalization. In this chapter, we will discuss double descent and grokking.

4.1 Double Descent

The classical problem in machine learning is to find a special point, often referred to as the "sweet spot", where the loss function is minimized on both training and test datasets [3], this point is called the "sweet spot" as shown in Fig. 4.1. For a long time, it was believed that this point represents the global optimum because moving further from this point typically leads to overfitting, where the loss function rapidly increases.This believe is based on the **bias-variance trade-off**, which suggests that increasing model complexity reduces bias (underfitting) but increases variance (overfitting). However, recent Nakkiran et al. [22] discovery has shown a surprising phenomenon: if we continue training beyond this peak, the error will start to decrease again, and we will "descend" to another minimum, denoted as $L(\mathcal{H})$. This phenomenon is known as **double descent**.

Analysis of the Graph $L(\mathcal{H})$ in Figure 4.1 illustrates two scenarios:

- **(a) U-shaped curve:** Represents the classical theory of machine learning.
- **(b) Double descent curve:** Shows the double descent phenomenon.

Belkin et al. [3] defines model capacity as the number of parameters required to define a function, which could be the number of features or the number of hidden neurons in a neural network.

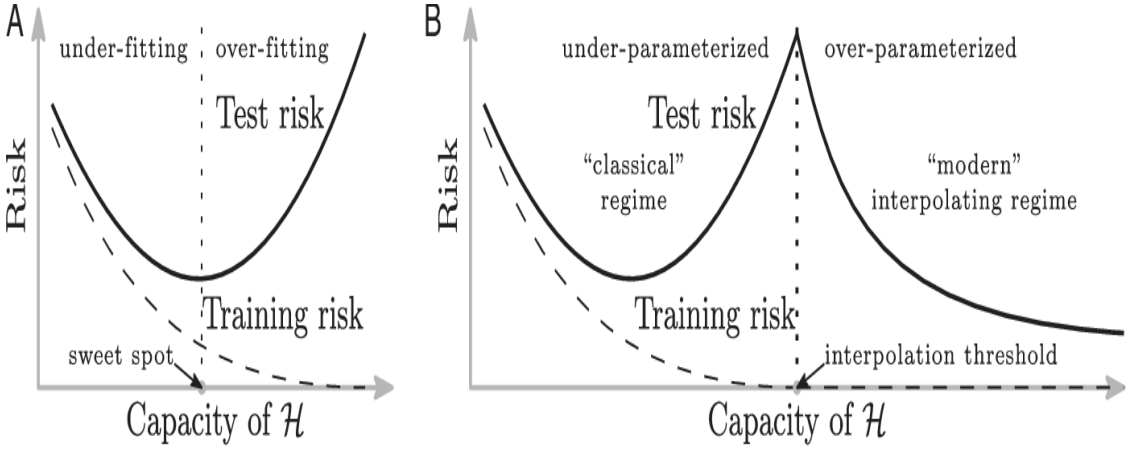


Figure 4.1: (A) U-shaped curve, classical theory. (B) Double descent curve. where $\mathcal{H}$ is the function class for specific neural network architecture [3].

Let's consider the point on graph (B) marked as the interpolation threshold. This is the maximum of the loss function, which is achieved when the number of objects in the sample N is equal to the number of features $\mathcal{H}$ [3]. This point divides the graph into two regions: under-parameterized and over-parameterized. In our study, we will focus on the latter – over-parameterized neural networks [3].

4.1.1 Double Descent Phenomenon.

Double Descent Phenomenon indicates that complex models can generalize well even beyond the sweet point where classical theory predicts overfitting. Neural networks may implicitly regularize themselves during training, possibly due to the nature of stochastic gradient descent, which might favor solutions that generalize better, even in over parameterized models.The noise introduced by Stochastic Gradient Decent might help in finding flatter minima in the loss landscape, which are associated with better generalization [41].

4.1.2 Epoch-wise Double Descent.

Epoch-wise Double Descent is a phenomenon similar to what we discussed earlier, but instead of plotting the complexity of the model on the x-axis, we plot the number of epochs. Everything mentioned earlier still holds true for Epoch-wise Dobble Decent [22], where test error initially decreases, then increases due to overfitting, and finally decreases again as the model continues to improves again as the model transitions to better generalization.

4.2 Grokking

Grokking is a fascinating and an interesting phenomenon observed in neural networks. It has been discovered with Power et al. [26], who noticed that neural networks could undergo a process where, after overfitting a training dataset and performing poorly on unseen data, they would suddenly begin to generalize exceptionally well with continued training.

4.2.1 What Happens During Grokking?.

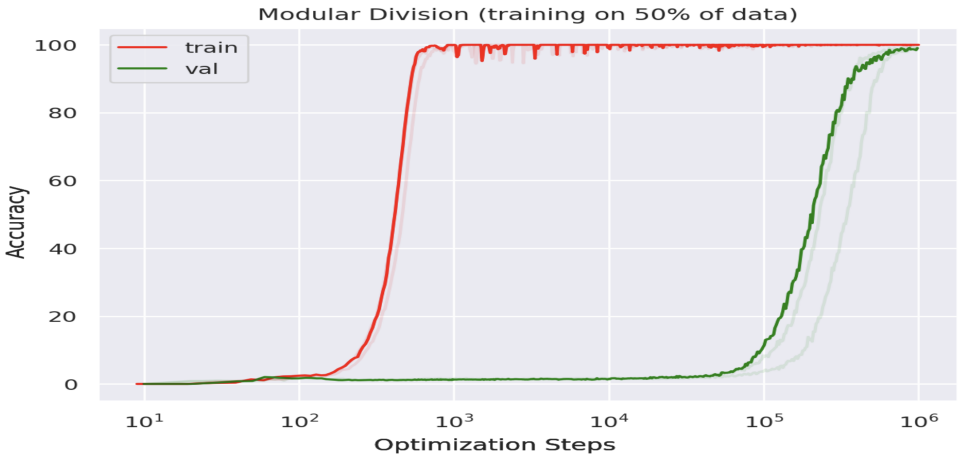


Figure 4.2: This figure illustrates the grokking phenomenon , where the x-axis represents the number of optimization steps on a logarithmic scale, and the y-axis represents accuracy. The red line represent the training accuracy and the green line represents the validation accuracy [26].

Initial Overfitting: During the initial training stages, the model memorizes the training data perfectly, achieving 100% training accuracy. However, it fails to generalize to the validation set, resulting in poor validation accuracy [26].

Sudden Generalization: After a significantly extended training period, the model undergoes an unexpected transition. It not only fits the training data but also generalizes well to the validation set, achieving near-perfect validation accuracy [26].

Power et al. [26] used synthetic datasets based on binary operation tables, such as addition, multiplication, composition of permutations, and modular arithmetic. These datasets are deterministic and noise-free, offering a clear environment to observe grokking . While grokking is observed on synthetic datasets.Experiments with noisy data show that grokking becomes more difficult with increased noise Power et al. [26].

Training Data Fraction: The fraction of training data impacts grokking with a larger fraction of the training data, grokking happens more quickly and reliably [26].

Weight Decay: Adding weight decay to the optimization process accelerates the grokking phenomenon. This regularization technique helps prevent the network from overfitting and encourages the discovery of simpler, more generalizable solutions [26].

4.3 Grokking Literature Review

This literature review explores various theories that seek to explain the mechanics of grokking, focusing on the existence of generalization solutions, the direction that separates memorization and generalization solutions, and the force driving the transition from memorization to generalization.

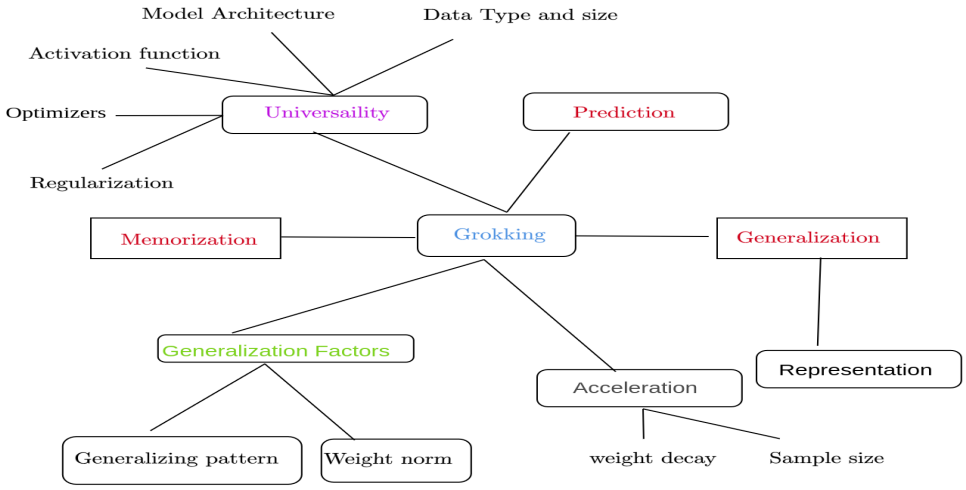


Figure 4.3: Grokking Literature map.

A key question is why generalization happens at all. Experts agree that learning good representations of data is essential [16] and others [26, 6, 25] suggest that a model's ability to create useful internal data representations is key to generalization . This means the model can understand patterns in the data rather than just memorize it. Generalization can be seen as moving through different areas in the model's parameters, with some areas linked to memorization and others to generalization. Different theories suggest various ways to measure this shift:

Weight Norm: The size of the model's parameters can show if the model is memorizing or generalizing. Keeping the weight norm small can prevent memorization [16].

Generalization and Memorization Pattern Formation : The way patterns form over time during training can help separate between memorization and generalization [7].

Understanding what drives the shift from memorization to generalization is important. Some ideas include:

Weight Decay: This technique penalizes large weights, pushing the model towards simpler, more generalizable solutions [16].

Optimization Process: The training process itself, especially gradient descent, may naturally lead the model towards generalization [23, 7].

Prediction By analyzing the loss function during training, researchers can predict and avoid grokking. Using the Fourier transform of the early loss curve can detect signs of grokking, but this method might not work for all types of neural networks and tasks [16, 13].

Universality Studies also look at how universal across various architecture and data sets as shown in Table (4.3). Non-algorithmic Datasets: Liu et al. [17] has shown that grokking can also happen with MINST data, not just algorithmic data.

Research Paper	Architecture	Category	Dataset Description
Power et al. [26]	Transformer	Algorithmic	Problems of the form $a \circ b = c$ where $\circ$ is a binary operation and $''a'',''b'',''\circ'',''='$ and $''c''$ are tokens.
Žunkovič and Ilievski [44]	Perceptron, Tensor Network, Rules-Based	Algorithmic	1D cellular automaton rule, 1D exponential and D-dimensional uniform ball.
Liu et al. [16]	MLP, Transformer	Algorithmic, Image class.	Addition modulo P, regular addition and MNIST.
Nanda et al. [23]	Transformer	Algorithmic	Addition modulo P.
Liu et al. [17]	MLP, LSTM, GCNN	Algorithmic, Image class., Language, Molecules	Regular addition, MNIST, IMDb dataset and QM9.
Merrill et al. [19]	MLP	Algorithmic	Parity prediction and operations modulo prime.
Davies et al. [7]	Transformer	Algorithmic	Addition modulo P.
Golechha [8]	MLP, Transformer, PolyNet	Algorithmic	Parity prediction task.
Murty et al. [21]	Transformer	Language	Question formation, tense-inflection and bracket nesting.
Liu et al. [16]	MLP	Algorithmic	XOR, S4 group operation and bitwise XOR.
Varma et al. [34]	Transformer	Algorithmic	Similar to Power et al. [26].

Table 4.1: Summary of datasets and architectures in which the grokking phenomenon has been studied. This table is taken from [20].

From our literature review, we discovered that certain model architectures, particularly Recurrent Neural Networks (RNNs) have not been studied in the context of grokking and (LSTM) has been studied by Liu et al. [17] on QM9 datasets . Our research aims to address this gap by focusing on RRNs and LSTM and examining their behavior under various regularization and optimization techniques on mod addition task from [26]. We will compare our findings with the results from previous studies by Power et al[26] and Liu et al. [16].

5. Methodology and Experimental Results

In this chapter, we present the results of our experiments and analyze the grokking behavior of different neural network architectures, optimizers, and regularization methods. We focus on the key metrics of training loss, validation loss, and accuracy to identify instances of grokking.

5.0.1 Experimental Design. To investigate grokking, we will use a dataset generated for a modular addition task. The dataset will consist of pairs of integers and their sums modulo a prime number. This task is chosen for its simplicity and its use in previous grokking studies [26].

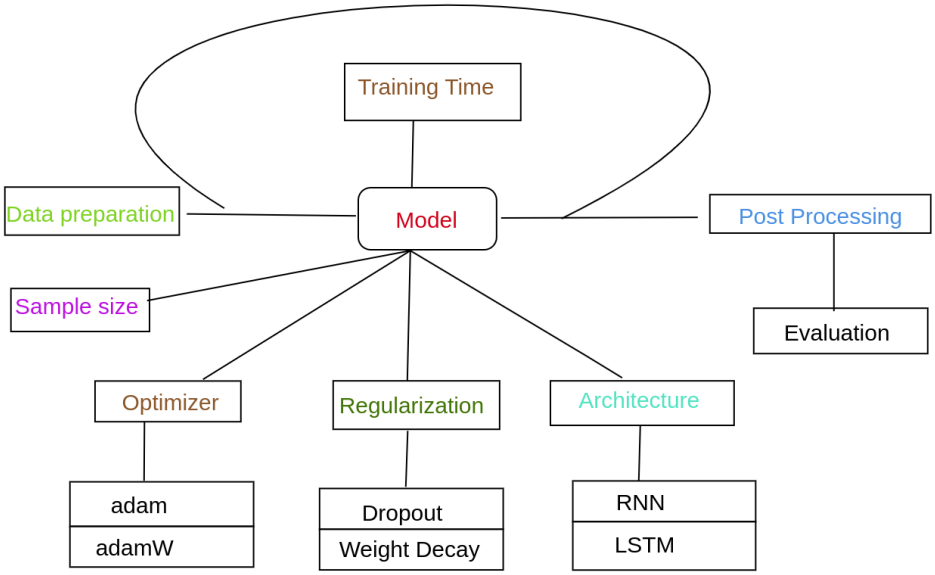


Figure 5.1: Grokking Experimental Design.

Impact of Optimizers and Regularization. We examine how different regularization methods impact grokking behavior.Then We analyze the effects of L2 regularization using AdamW 3.1, Dropout 3.1.1, and without regularization. The results exploring the effect of dropout can be found in the appendix in Figure 7.3 and Figure 7.3b.

Evaluation Metrics. The performance of the models will be evaluated using the following metrics:

- **Training Los**: Cross-entropy loss measured during training.
- **Validation Loss**: Cross-entropy loss measured on the validation set.
- **Accuracy**: The percentage of correctly predicted outputs in both training and validation sets.

Training Procedure.
For each possible combination of model architectures, optimizers, and regularization methods.

Analysis.
Post-training, we will analyze the training and validation loss curves, and accuracy to identify the grokking. Specifically, we will look for instances where the validation accuracy suddenly improves after a period of overfitting , which is indicative of grokking. We will compare these trends across different architectures and training configurations to understand how they impact the grokking phenomenon. Monitoring weight norms and key evaluation metrics during training provides valuable insights into the transition from overfitting to generalization, helping to better understand and optimize neural network training for improved generalization performance.

Tools and Libraries.
The experiments will be implemented using PyTorch for building and training the neural network models. Data preparation and post processing (visualization) will be done using NumPy and Matplotlib our code is adapted from Liu et al. [17].

5.0.2 Architecture and Training Set Size. Model Architectures.

We will investigate the grokking phenomenon using Two types of neural network architectures: Recurrent Neural Networks (RNN) in Section 2.3.1 and LSTM in Section 2.3.2.

Sample Size. Lastly, we explore how varying the sample size (30%, 50%, 70%) affects grokking behavior. We analyze whether smaller sample sizes lead to more pronounced grokking.

5.0.3 Weight Decay Experiments .

We utilized an LSTM model designed to perform modular addition. The model was trained on 50% of the dataset . Our baseline configuration involved an initial weight decay of 0.01, as employed in earlier experiments.

Our experiments followed a structured approach:

Baseline Experiment with No Weight Decay.

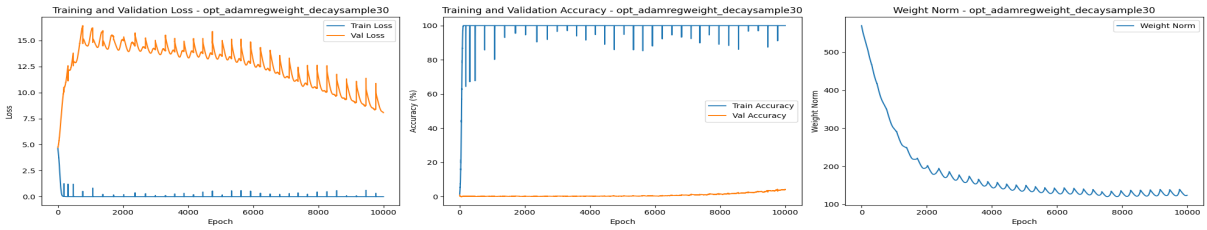
We first trained the LSTM model with weight decay set to zero throughout the training process. This served to confirm whether removing weight decay entirely would prevent grokking.

Dynamic Weight Decay Adjustment.

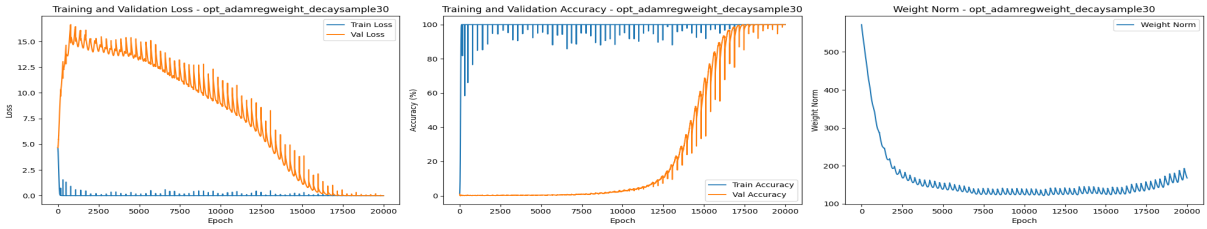
After confirming the baseline results, we ran a series of experiments where weight decay was initially set to 0.01 and then removed at specific training epochs or performance thresholds. Further, we ran experiments where weight decay was initially set to 0 and then turned on weight decay at later training epochs . This approach aimed to determine critical phases during training when weight decay impacts the occurrence of grokking.

5.1 Role of architecture and Training Set Size

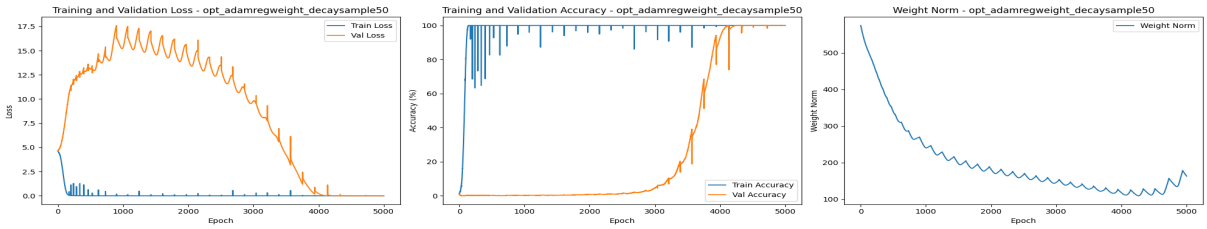
5.1.1 Grokking in LSTM Models with Varying Training Data Fractions.



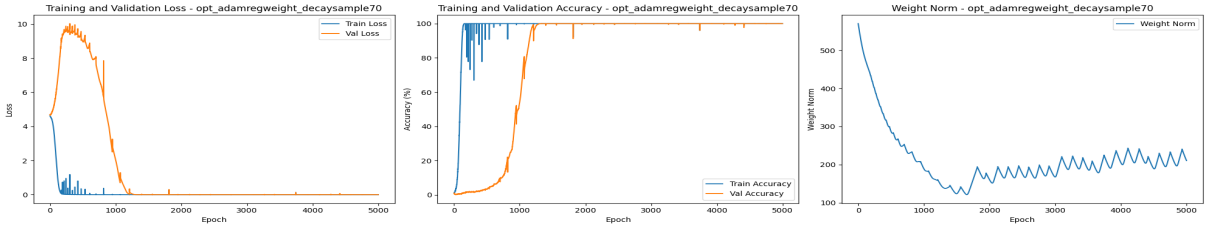
(a) LSTM model trained on 30% of the data.



(b) Training and validation loss curves for the LSTM model trained on 30% of the data for an extended period.



(c) Training and validation loss curves for the LSTM model using 50% of the data.

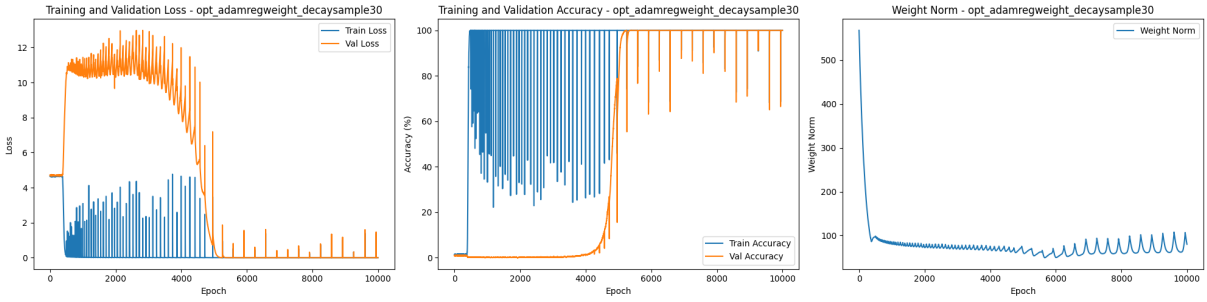


(d) Training and validation curves for the LSTM model trained on 70% of the dataset.

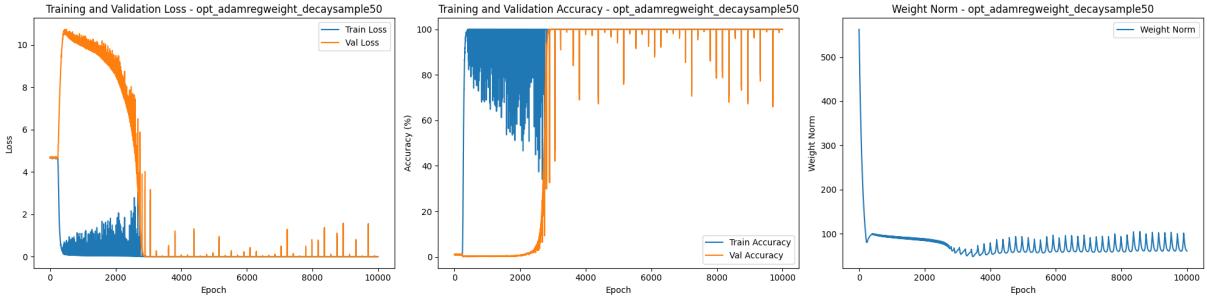
Figure 5.2: Training and validation curves for the LSTM models at 30%, 50%, and 70% training data fractions.

Observation. LSTM models in Figure 5.2 exhibit the grokking phenomenon. We observe that the validation accuracy undergoes a sudden improvement after a prolonged period of training, indicating grokking.

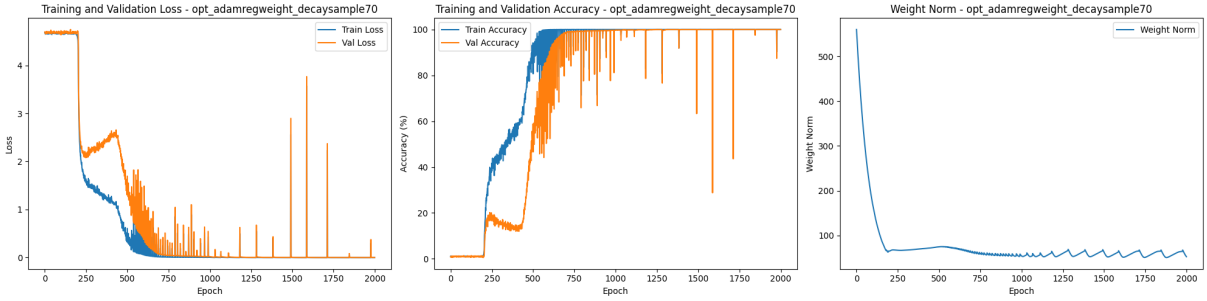
5.1.2 Grokking in RNNs Models with varying Training Data Fractions.



(a) Training and validation loss curves for an RNN model trained on 30% of the dataset.



(b) Training and validation loss curves for the RNN model trained on 50% of the data.



(c) Training and validation loss curves for the RNN model trained on 70% of the data.

Figure 5.3: Training and validation curves for the RNN models at 30%, 50%, and 70% training data fractions.

Observation. RNNs in Figure 5.3 exhibit the grokking phenomenon. We observe that the validation accuracy undergoes a sudden improvement after a prolonged period of training, indicating grokking.

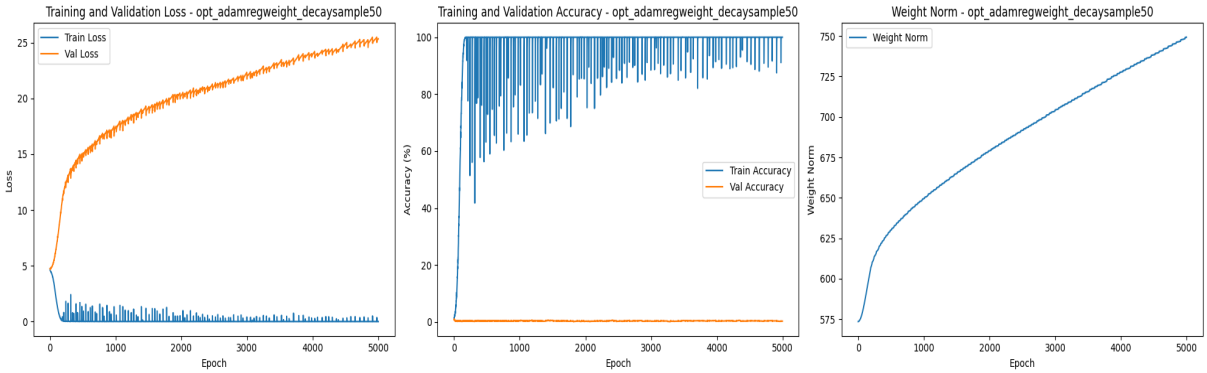
5.2 Experiment: The Role of Regularization

Experimental Setup:

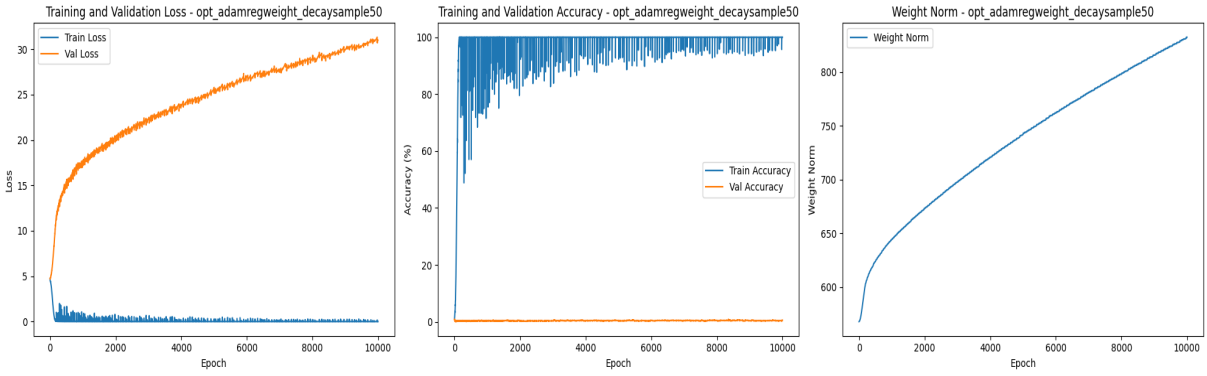
We utilized an LSTM model designed to perform modular addition. The model was trained on 50% of the dataset, chosen to provide more noticable grokking with reasonable number of optimization steps while observing the effects of training dynamics on grokking. Our baseline configuration involved an initial weight decay of 0.01, as employed in earlier experiments as shown in Table 7.2.

5.2.1 Baseline Experiment.

In our first experiment, we did not use weight decay during training. Without weight decay, the model starts to overfit the training data. However, it did not exhibit grokking as we expected. This confirmed our expectation that weight decay plays a role in controlling the weight norm, thereby reducing overfitting and enhancing the model's ability to generalize to unseen datasets as shown in Figure 5.4b below.



(a) Training and validation loss curves for the LSTM model with weight decay set to zero.



(b) Training and validation loss curves for the LSTM model with weight decay set to zero, trained for an extended period to ensure that there will be any delay grokking .

Figure 5.4: Training and validation loss curves for the LSTM model with weight decay set to zero. (a) Shows the model overfitting the training data but failing to exhibit grokking without weight decay. (b) Shows the same model trained for an extended period.

5.2.2 Experiment: Remove weight decay during training .

We aimed to identify the effects of turning off weight decay at different epochs during the early and late phases of training.

Training Dynamics for Early Phase (Epoch 200): Removing weight decay early in the training process leads to overfitting and poor generalization. Validation accuracy remains low, highlighting the challenge of achieving generalization without weight decay. The weight norm continues to increase steadily after weight decay is removed.

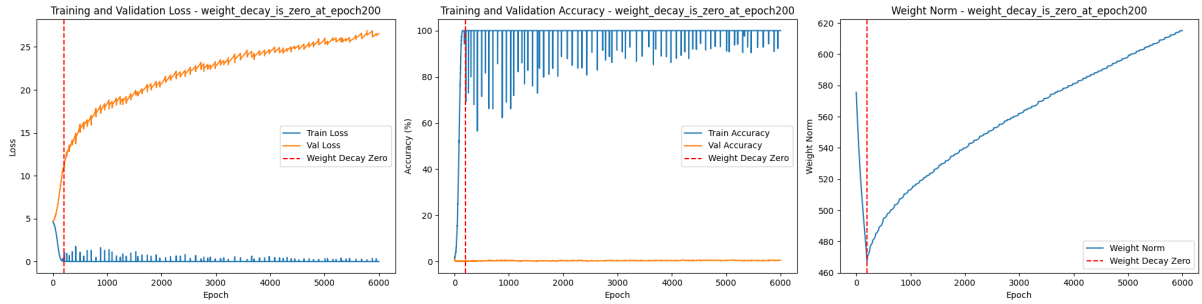


Figure 5.5: Training and validation dynamics with weight decay set to zero at epoch 200. (Left) Training and validation loss over epochs. (Center) Training and validation accuracy over epochs. (Right) Weight norm over epochs. The red dashed line marks the point at which weight decay is set to zero , more result can be found in the Appendix in Figure 7.5.

Training Dynamics for Late Phase (Epoch 4270): Removing weight decay at a late phase has no significant impact on the model's generalization performance once grokking has occurred. Validation accuracy remains high, demonstrating that the model's generalization capability is robust to changes in regularization.

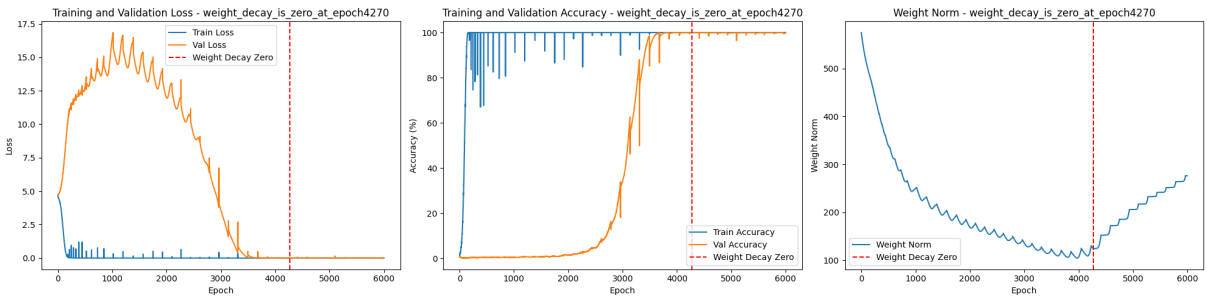


Figure 5.6: Training and validation dynamics with weight decay set to zero at epoch 4270. (Left) Training and validation loss over epochs. (Center) Training and validation accuracy over epochs. (Right) Weight norm over epochs. The red dashed line marks the point at which weight decay is set to zero , more result can be found in the Appendix in Figure 7.6.

Observation. Once the model has exhibited grokking and stabilized its generalization performance, the removal of weight decay does not affect the model as shown in Figure 7.6.

5.2.3 Experiment: Introduce weight decay during training .

We aim to identify the effects of turning off weight decay from the beginning and then reapplying it at different epochs, with either the same or increased weight decay values.

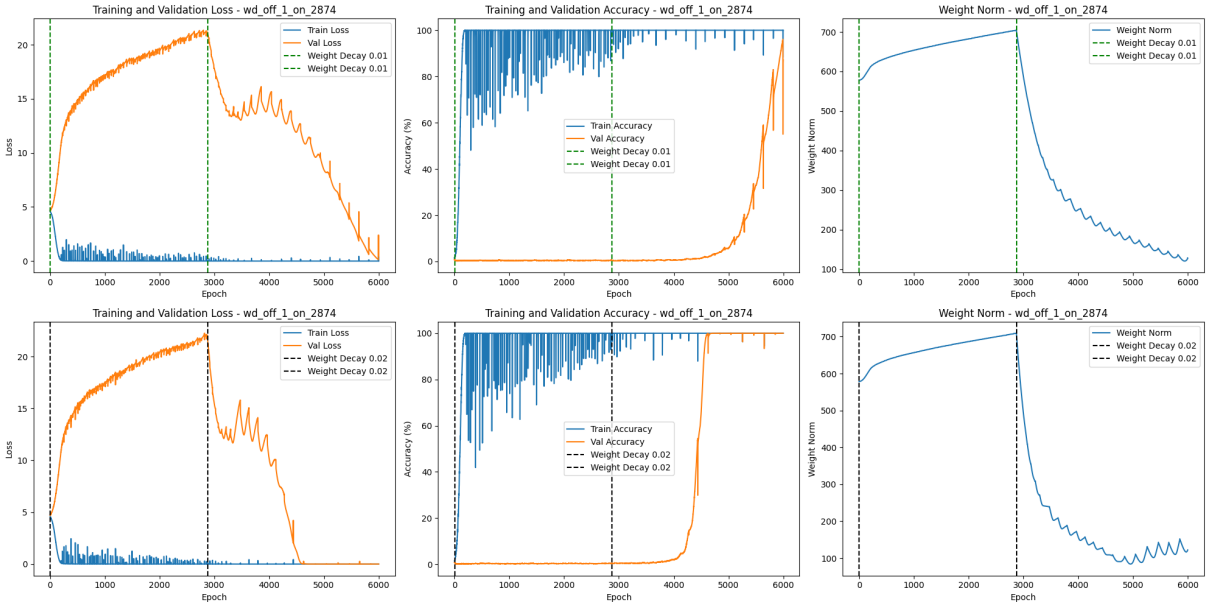


Figure 5.7: This figure presents the training and validation loss, accuracy, and weight norm dynamics for the LSTM model trained on the same dataset with varying weight decay scenarios. Each row represents a different experimental setting where weight decay was dynamically adjusted. Each column provides insights into different metrics. The green and black dashed lines indicate the points where weight decay of 0.01 and 0.02, respectively, were applied or removed.more result can be found in the Appendix in Figure 7.7.

Initial Removal of Weight Decay.

Training Dynamics: Starting training without weight decay allows the model to initially fit the training data without regularization constraints, leading to high training accuracy and overfitting, as indicated by high validation loss.

Reapplication of Weight Decay at a Later Phase.

Same Value Reapplication: Stabilizes and improves validation performance, reducing validation loss and constraining weight norm. Double Value Reapplication: Provides a stronger regularization effect, leading to a pronounced reduction in weight norm and improved validation accuracy.

Key Observation.

Earlier Occurrence of Grokking with Higher Weight Decay: When weight decay is reapplied at double the previous value, the grokking phenomenon occurs earlier compared to reapplying the same weight decay value. This suggests that stronger regularization accelerates the transition from overfitting to generalization, leading to earlier grokking. We find that even when weight decay is set off initially and only introduced late in training, we still observe grokking which is a novel discovery.

5.3 Summary of Results

In our experiments, we investigated the grokking phenomenon using two types of neural network architectures: Recurrent Neural Networks (RNNs) 2.3.1, and Long Short-Term Memory (LSTM) networks 2.3.2. We examined the impact of different regularization methods (L2 regularization, dropout, and no regularization) on grokking behavior. Additionally, we explored how varying the sample size (30%, 50%, and 70%) affects grokking.

Finding RNN and LSTM Behavior.

Observation: Initially, LSTM 7.8b and RNNs 7.8a models exhibited faster generalization due to their smaller initial weight norms .leading to a more gradual improvement in both training and validation accuracy. Small initial weights may prevent immediate overfitting, leading to a more gradual improvement in both training and validation accuracy as shown in Figure 7.8b. However, to observe grokking similar to that in Transformer employed by Power et al. [26] as shown in Table 7.1, we scaled the weight norms in these models. The initialization scale was motivated through the finding of Liu et al. [17]. After scaling the weight norms using the same method of Liu et al. [17], LSTM and RNNs models exhibited grokking behavior similar to Transformers, suggesting that initial weight norm scaling can significantly impact the grokking phenomenon.

Relation to Weight Initialization and Regularization:

- **Small Initializations:** Initially, LSTM 7.8b and RNN 7.8a models had smaller initial weight norms, which led to faster generalization. By scaling the weight norms, we introduced a larger initial weight norm, resulting in grokking behavior that includes long initial overfitting period followed by a rapid improvement in validation accuracy.

- **Explicit Regularization:** The use of weight decay was critical for observing grokking in these models. Without weight decay, grokking was not observed, indicating that explicit regularization was necessary to guide these models towards generalizing solutions.

  Impact on Grokking: This penalty helps the model escape overfitting regions by gradually reducing the weight norms, pushing the model towards the regions where generalization is more likely. Weight decay can thus shorten the overfitting period and lead to quicker generalization.As more decay we add the faster grokking can happens. Training without weight decay leads to poor generalization due to overfitting as shown in Figure 5.2.1. Removing weight decay early also results in poor generalization because the model starts to overfit in the later stages of training as shown in Figure 5.5. Applying weight decay only in the later stages of training can still lead to effective generalization and grokking as shown in Figure 5.7, which is a novel discovery.

- **Implicit Regularization Mechanisms:** Implicit regularization can occur through factors like optimizer choice and learning rate schedules, even without explicit penalties like weight decay. However we couldn't observer grokking with out explicit regularization.

5.4 Finding 1: Impact of Sample Size on Grokking

Smaller Sample Sizes (30% and 50%): Smaller sample sizes led to more pronounced grokking behavior. Training with 30% of the data might show the model overfitting quickly but then grokking after a long training period, as shown in Figure 5.2 and Figure 5.3. This aligns with the Liu et al. [17] findings, where they observed that using smaller training sizes led to more noticeable grokking. For instance, when they reduced the MNIST training set from 60,000 to 1,000 samples, they observed similar grokking behavior [17].

Larger Sample Size (70%): With 70% of the data, grokking was still observed but was less pronounced compared to smaller sample sizes. Similarly, in the Liu et al. [17] experiments, larger training sizes resulted in less pronounced grokking. For instance, in their sentiment analysis on the IMDb dataset, using more training data led to a smoother and faster transition from overfitting to generalization [17].

Implication: Smaller datasets tend to make the grokking phenomenon more pronounced. The model initially overfits due to the limited data but then finds generalizing solutions after extended training, as shown in Figure 5.2b. This matches Liu et al. [17] results, where smaller datasets caused a longer period of overfitting before the model started to generalize. Liu et al. [17] explained this as a result of differences between the training and test loss landscapes, which delays generalization.

5.4.1 Conclusion. Our findings about the impact of training set size on grokking are similar to what Liu et al. [17] found, even though they used different types of datasets and training configurations.

5.5 Finding 2: Impact of Weight Decay on Grokking

- **With Weight Decay:** Weight decay played a crucial role in facilitating grokking. Without weight decay, models tended to overfit without showing signs of grokking.
- **Dynamic Adjustment of Weight Decay Behavior:** Dynamically adjusting weight decay, such as turning it off after reaching specific accuracy thresholds, can impact training dynamics as shown in Figure 7.5 and Figure 7.7.

  Impact on Grokking: Training without weight decay leads to poor generalization due to overfitting as shown in Figure 5.2.1. Removing weight decay early also results in poor generalization because the model starts to overfit in the later stages of training as shown in Figure 5.5. Applying weight decay only in the later stages of training can still lead to effective generalization and grokking as shown in Figure 5.7, which is a novel discovery.

- **Prolonged Training:** Extended training allows the explicit regularization mechanism (weight decay) to take effect, helping the model escape the overfitting region and discover patterns that generalize well to the validation set.

6. Conclusion and Further Work

6.0.1 Summary of Findings.

Our experiments confirmed that weight decay, sample size, and extended training are critical factors influencing the grokking phenomenon in neural networks. Different neural network architectures exhibit distinct grokking behaviors, and adaptive optimizers like Adam and AdamW, along with proper regularization, play significant roles in facilitating grokking.

These findings align with finding of Liu et al. [17] and Power et al. [26] on the role of both implicit and explicit regularization in guiding neural networks towards generalization. The interplay between weight initialization, regularization, and sample size is important in understanding grokking. Large weight initializations and appropriate regularization (both explicit and implicit) are key factors that influence when and how grokking occurs. By manipulating these factors, we can better control the training dynamics and improve the generalization performance of neural networks.

6.0.2 Future Research Directions.

In our future research, we aim to explore additional neural network architectures to further understand the phenomenon of grokking. Specifically, we plan to investigate new architectures, such as Mamba [2] and XLSTM [9], to determine if these architectures exhibit similar grokking behaviors and to identify any unique patterns that may emerge. Additionally, we want to understand the conditions under which explicit regularization, such as weight decay, is necessary to exhibit grokking. By finding out when these specific techniques are needed, we can better manage and improve the training process for neural networks, ensuring optimal performance and generalization.

6.0.3 Concluding Remarks.

The discovery of grokking by Power et al. [26] reveals that under specific conditions neural networks can transition from overfitting to exceptional generalization. This finding challenges traditional views on model training and overfitting. Understanding the grokking phenomenon could lead to valuable insights into the generalization behavior of neural networks.

References

[1] Maksym Andriushchenko, Francesco D'Angelo, Aditya Varre, and Nicolas Flammarion. Why do we need weight decay in modern deep learning? *arXiv preprint arXiv:2310.04415*, 2023.

[2] Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

[3] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[4] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade: Second Edition*, pages 421–436. Springer, 2012.

[5] Rohitash Chandra, Ayush Jain, and Divyanshu Singh Chauhan. Deep learning via lstm models for covid-19 infection forecasting in india. *PloS one*, 17(1):e0262708, 2022.

[6] Bilal Chughtai, Lawrence Chan, and Neel Nanda. A toy model of universality: Reverse engineering how networks learn group operations. In *International Conference on Machine Learning*, pages 6243–6267. PMLR, 2023.

[7] Xander Davies, Lauro Langosco, and David Krueger. Unifying grokking and double descent. *arXiv preprint arXiv:2303.06173*, 2023.

[8] Satvik Golechha. Progress measures for grokking on real-world datasets. *arXiv preprint arXiv:2405.12755*, 2024.

[9] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[11] Yufei Huang, Shengding Hu, Xu Han, Zhiyuan Liu, and Maosong Sun. Unified view of grokking, double descent and emergent abilities: A perspective from circuits competition. *ML Safety Workshop NeurIPS 2022*, 2024.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[13] Pascal Notsawo Jr, Hattie Zhou, Mohammad Pezeshki, Irina Rish, and Guillaume Dumas et al. Predicting grokking long before it happens: A look into the loss landscape of models which grok. *arXiv preprint arXiv:2306.13253*, 2023.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[15] Jake Lever, Martin Krzywinski, and Naomi Altman. Points of significance: model selection and overfitting. *Nature methods*, 13(9):703–705, 2016.

[16] Ziming Liu, Ouail Kitouni, Niklas S Nolte, Eric Michaud, Max Tegmark, and Mike Williams. Towards understanding grokking: An effective theory of representation learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 34651–34663, 2022.

[17] Ziming Liu, Eric J Michaud, and Max Tegmark. Omnigrok: Grokking beyond algorithmic data. In *The Eleventh International Conference on Learning Representations*, 2022.

[18] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *ICLR*, 2019.

[19] William Merrill, Nikolaos Tsilivis, and Aman Shukla. A tale of two circuits: Grokking as competition of sparse and dense subnetworks. *arXiv preprint arXiv:2303.11873*, 2023.

[20] Jack Miller, Charles O'Neill, and Thang Bui. Grokking beyond neural networks: An empirical exploration with model complexity. *arXiv preprint arXiv:2310.17247*, 2023.

[21] Shikhar Murty, Pratyusha Sharma, Jacob Andreas, and Christopher D Manning. Grokking of hierarchical structure in vanilla transformers. *arXiv preprint arXiv:2305.18741*, 2023.

[22] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124003, 2021.

[23] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

[24] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Accessed: June 2, 2024.

[25] Google PAIR. Grokking machine learning. https://pair.withgoogle.com/explorables/grokking/, 2024. Accessed: 2024-05-31.

[26] Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

[27] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[29] Robin M Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv preprint arXiv:1912.05911*, 2019.

[30] Towards Data Science. Backpropagation step-by-step derivation. *Towards Data Science*, 2023. URL https://towardsdatascience.com/backpropagation-step-by-step-derivation-99ac8fbdcc28. Accessed: 2024-06-21.

[31] Abha Singh, Sumit Kushwaha, Maryam Alarfaj, and Manoj Singh. Comprehensive overview of backpropagation algorithm for digital image denoising. *Electronics*, 11(10):1590, 2022.

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[33] Ralf C Staudemeyer and Eric Rothstein Morris. Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.

[34] Vikrant Varma, Rohin Shah, Zachary Kenton, János Kramár, and Ramana Kumar. Explaining grokking through circuit efficiency. *arXiv preprint arXiv:2309.02390*, 2023.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[36] K You, M Long, J Wang, and MI Jordan. How does learning rate decay help modern neural networks? arxiv 2019. *arXiv preprint arXiv:1908.01878*, 1908.

[37] Aston Zhang, Zachary C. Lipton, and Mu Li. Dive into deep learning: Activation functions. https://classic.d2l.ai/chapter_multilayer-perceptrons/mlp.html#activation-functions, 2020. Accessed: June 2, 2024.

[38] Aston Zhang, Zachary C. Lipton, and Mu Li. Dive into deep learning. https://classic.d2l.ai/chapter_multilayer-perceptrons/mlp.html, 2020. Accessed: June 2, 2024.

[39] Aston Zhang, Zachary C. Lipton, and Mu Li. Dive into deep learning. 2020. Accessed: June 2, 2024.

[40] Aston Zhang, Zachary C. Lipton, and Mu Li. Adam. https://classic.d2l.ai/chapter_optimization/adam.html, 2020. Accessed: 2024-05-31.

[41] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

[42] Wenyi Zhang, Xiaohua Shen, Haoran Zhang, Zhaohui Yin, Jiayu Sun, Xisheng Zhang, and Lejun Zou. Feature importance measure of a multilayer perceptron based on the presingle-connection layer. *Knowledge and Information Systems*, 66(1):511–533, 2024.

[43] Zhenxun Zhuang. *Adaptive strategies in non-convex optimization*. PhD thesis, Boston University, 2022.

[44] Bojan Žunkovič and Enej Ilievski. Grokking phase transitions in learning local rules with gradient descent. *arXiv preprint arXiv:2210.15435*, 2022.

7. appendices

7.1 Notations and Setup

We use the same notation as Andriushchenko et al. [1], with modifications to suit our modular addition task.

Training Data Let $(\mathbf{x}_i, y_i)_{i=1}^n$ be the training inputs and labels.

- $\mathbf{x}_i \in \mathcal{D}$ represents the sequence of tokens (e.g., ['<sos>', 'x ', '+',' y ', 'mod' ,'=']).
- $y_i \in \mathbb{R}$ represents the result of the arithmetic operation (e.g., $(x + y) \mod 91$).
- y_i is one-hot encoded in $\mathbb{R}^C$, where C is the number of possible results modulo mod.

Hypothesis Class

- $h : \mathbb{R}^p \times \mathcal{D} \to \mathbb{R}^C$ represents the hypothesis class of neural networks.
- $\mathbf{w} \in \mathbb{R}^p$ denotes the parameters of the neural network.
- $h(\mathbf{w}, \cdot) : \mathcal{D} \to \mathbb{R}^C$ represents the network's predictions for inputs in $\mathcal{D}$.

Overparameterization Assumption The network is overparameterized, meaning it has more parameters than necessary and can achieve perfect training accuracy.

Training Loss

$$\mathcal{L}(\mathbf{w}) := \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, h(\mathbf{w}, \mathbf{x}_i)) \tag{7.1.1}$$

The training loss, where $\ell(\cdot, \cdot)$ is the cross-entropy loss function.

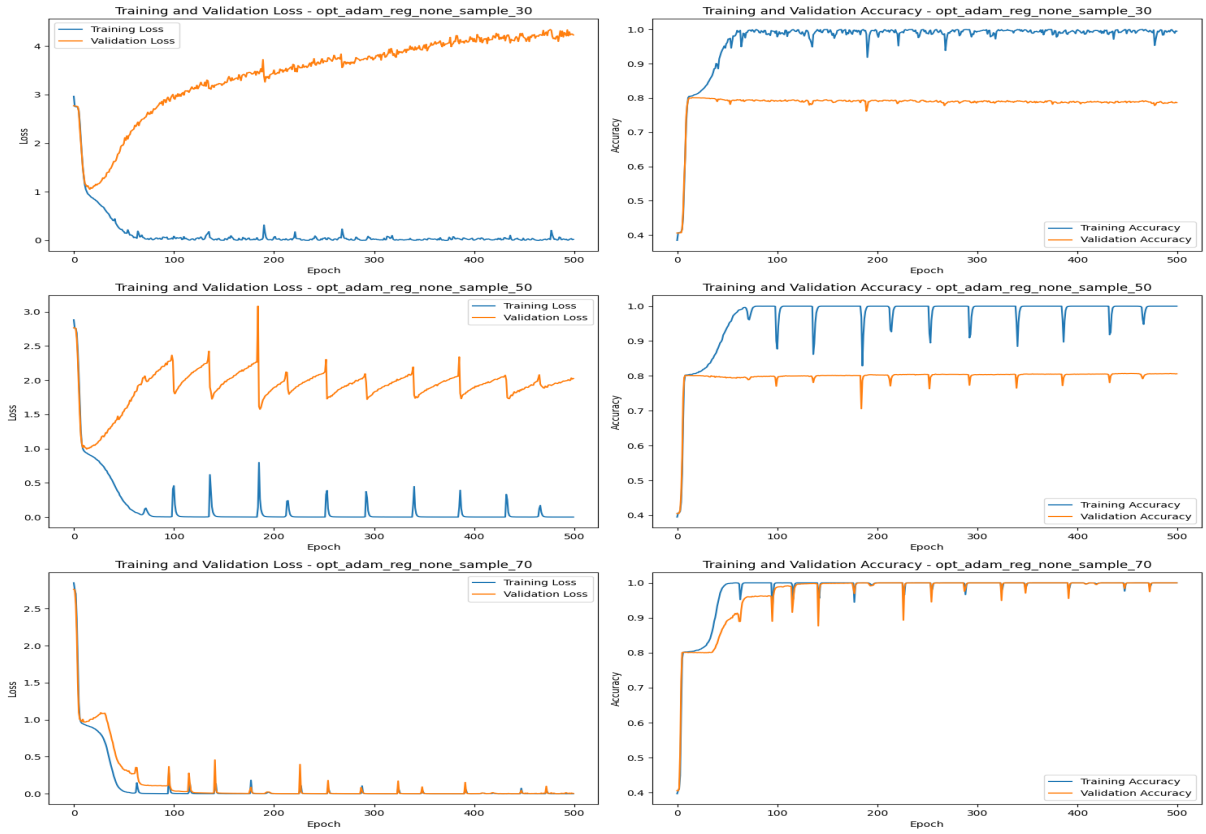
Parameter	Description	Numerical Value
num_layers	Number of Transformer blocks in the model.	1
d_vocab	Dimensionality of the vocabulary.	114
d_model	Dimensionality of the model.	128
d_mlp	Dimensionality of the hidden layer in the feedforward neural network.	512
d_head	Dimensionality of each attention head.	32
num_heads	Number of attention heads in the multi-head attention mechanism.	4
n_ctx	Context length considered in each Transformer block.	3
act_type	Type of activation function used in the Transformer block.	ReLU
use_cache	Boolean indicating whether cache mechanism is used.	False
use_ln	Boolean indicating whether Layer Normalization is used.	False

Table 7.1: Parameters for Transformer Class.

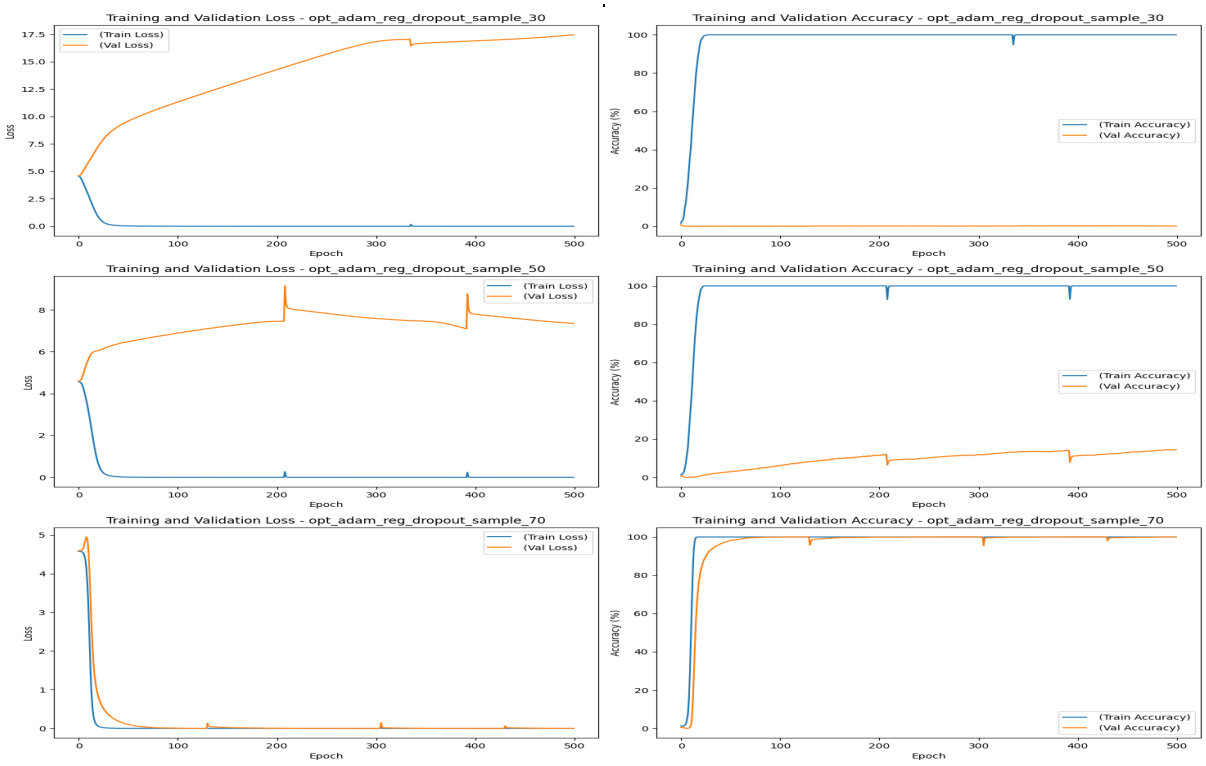
Model Hyperparameters	
Embedding Size	128 (Size of the embeddings)
Hidden Size	128 (Number of hidden units in the LSTM)
Dropout Rate	or 0.5 depending
Alpha	5.0 (Weight initialization scaling factor)
Training Hyperparameters	
Optimizer	AdamW
Regularizations	0,01Weight decay
Sample Sizes	0.5 (Fraction of the dataset used for training)
Number of Epochs	6000
Batch Size	32

Table 7.2: Model and Training Hyperparameters For LSTM.

Experiment Results

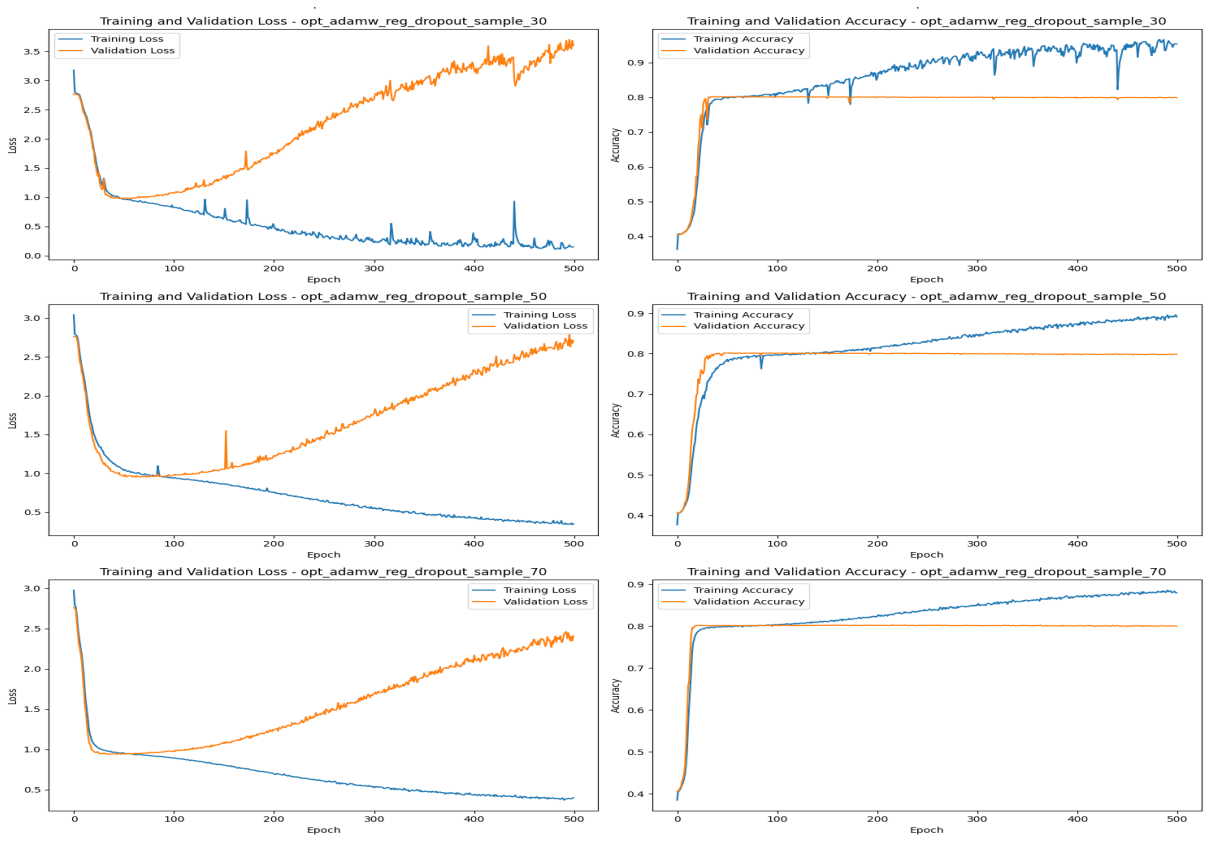


(a) Training and validation loss and accuracy for an RNN model trained with different sample sizes (30%, 50%, and 70%) using the Adam optimizer without regularization.



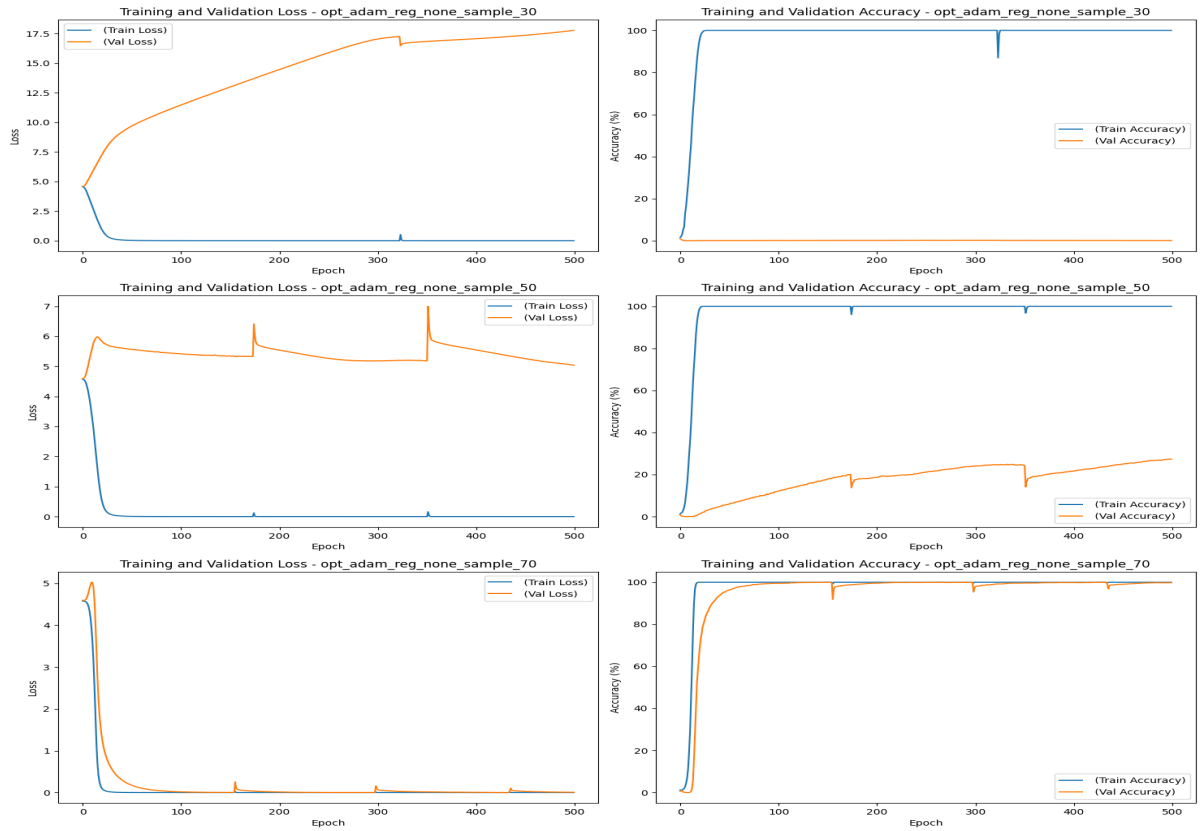
(b) Training and validation loss and accuracy for an RNN model trained with different sample sizes (30%, 50%, and 70%) using the Adam optimizer and dropout regularization.

Figure 7.1: Training and validation performance for RNN models trained with different sample sizes (30%, 50%, and 70%) using (a) the Adam optimizer without regularization and (b) the Adam optimizer with dropout regularization.

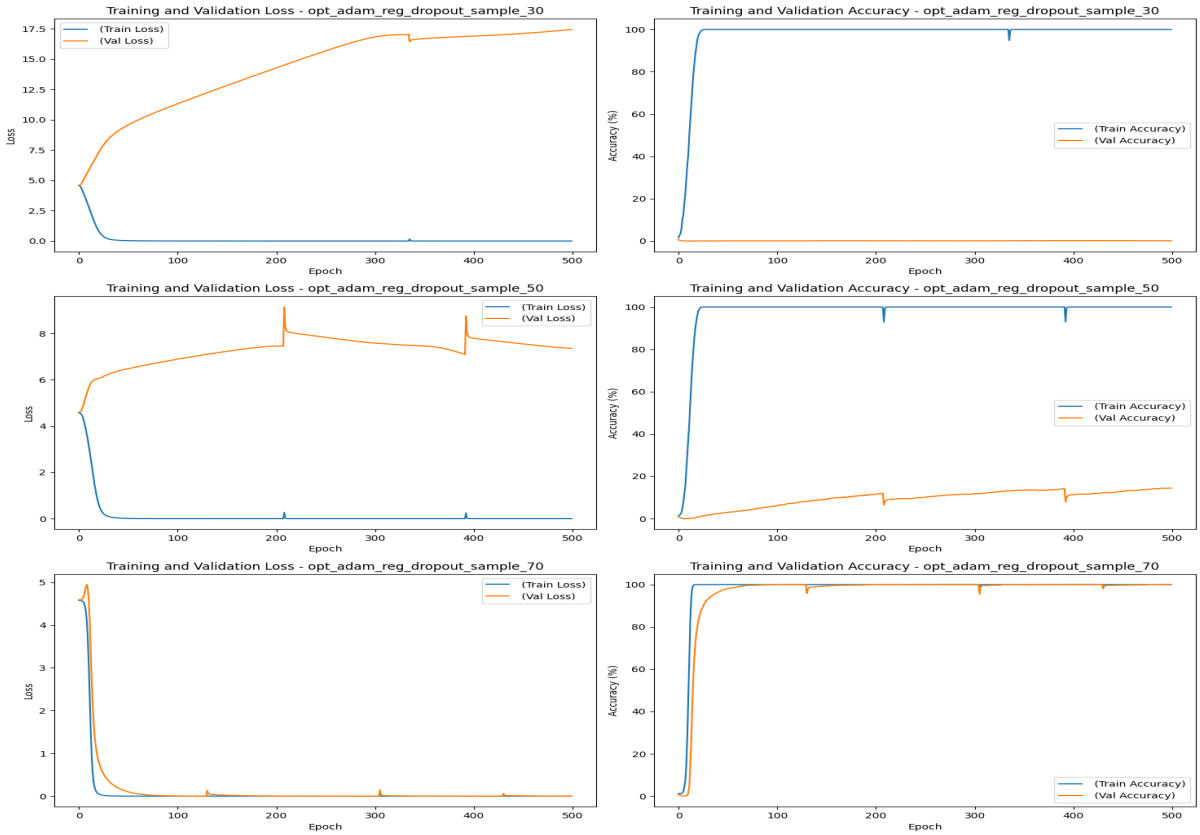


(a) Training and validation loss and accuracy for an RNN model trained with different sample sizes (30%, 50%, and 70%) using the AdamW optimizer and dropout regularization.

Figure 7.2: Training and validation performance for RNN models trained with different sample sizes (30%, 50%, and 70%) using the AdamW optimizer with dropout regularization.

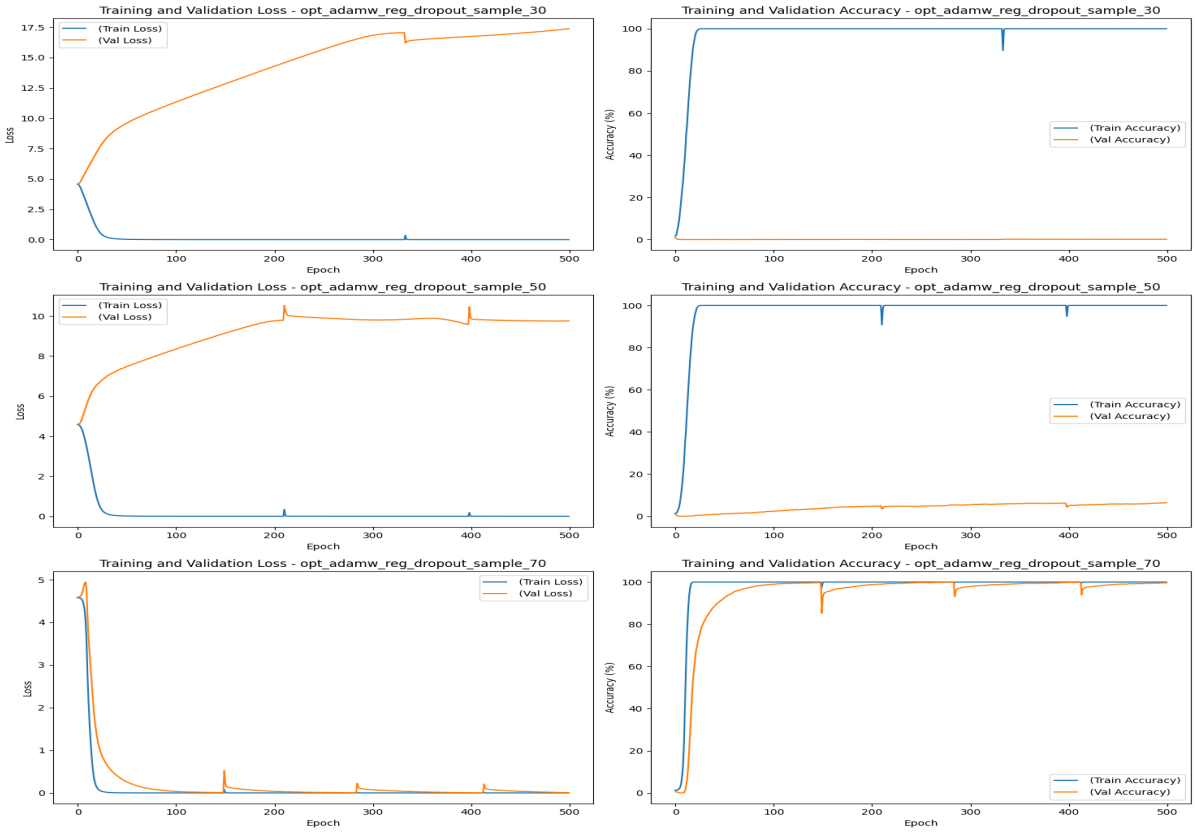


(a) Training and validation loss and accuracy for an LSTM model trained with different sample sizes (30%, 50%, and 70%) using the Adam optimizer without regularization.



(b) Training and validation loss and accuracy for an LSTM model trained with different sample sizes (30%, 50%, and 70%) using the Adam optimizer and dropout regularization.

Figure 7.3: Training and validation performance for LSTM models trained with different sample sizes (30%, 50%, and 70%) using (a) the Adam optimizer without regularization and (b) the Adam optimizer with dropout regularization.



(a) Training and validation loss and accuracy for an LSTM model trained with different sample sizes (30%, 50%, and 70%) using the AdamW optimizer and dropout regularization.

Figure 7.4: Training and validation performance for LSTM models trained with different sample sizes (30%, 50%, and 70%) using the AdamW optimizer with dropout regularization.

7.2 Weight Decay

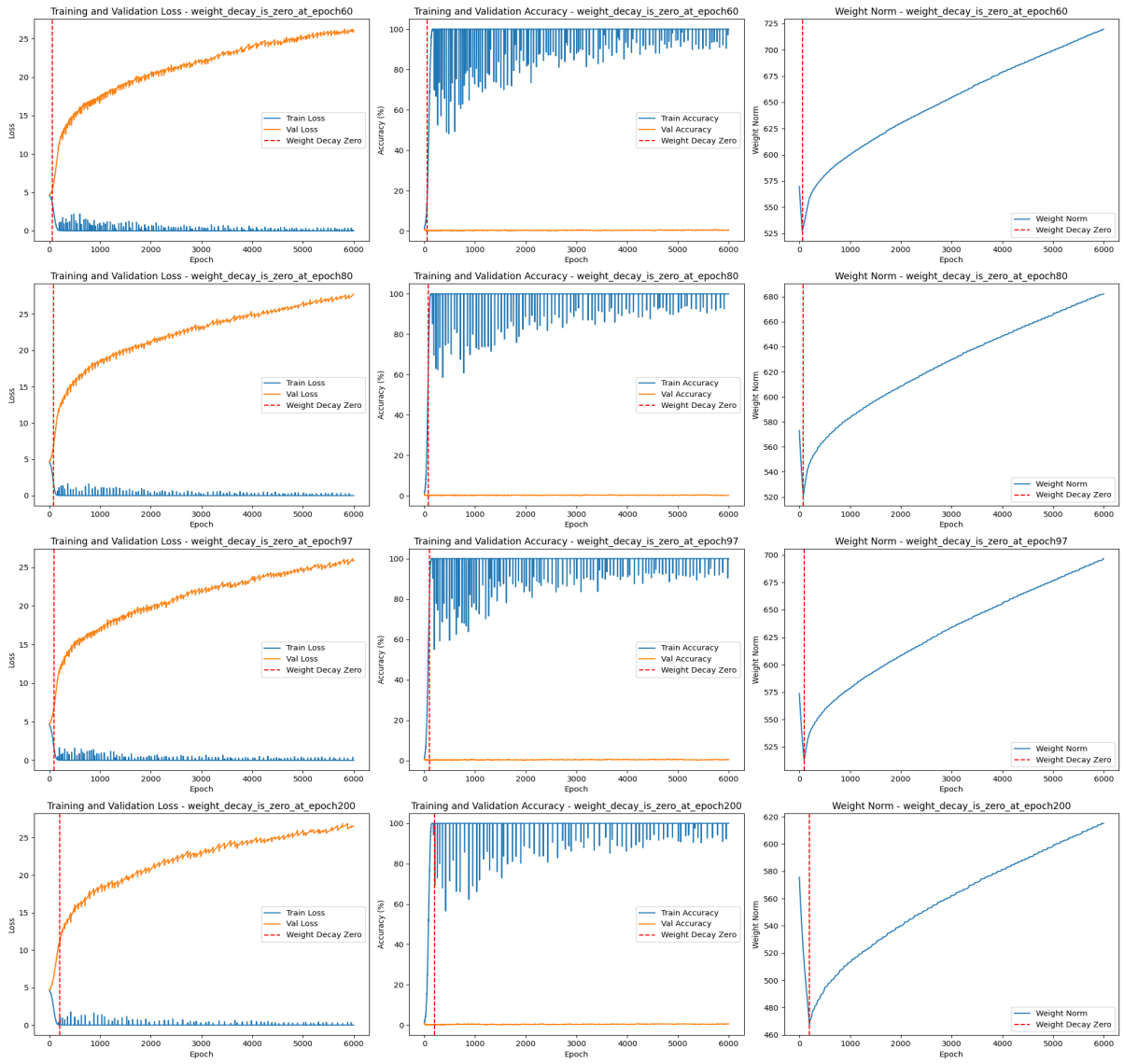


Figure 7.5: Training and validation loss, accuracy, and weight norm for the LSTM model trained on 50% of the dataset. Weight decay is initially applied and then set to zero at different epochs (60, 70, 97, and 200). Each row represents a different experiment with weight decay removed at a specific epoch. Each column provides insights into different metrics. The red dashed lines indicate the epochs where weight decay was removed.

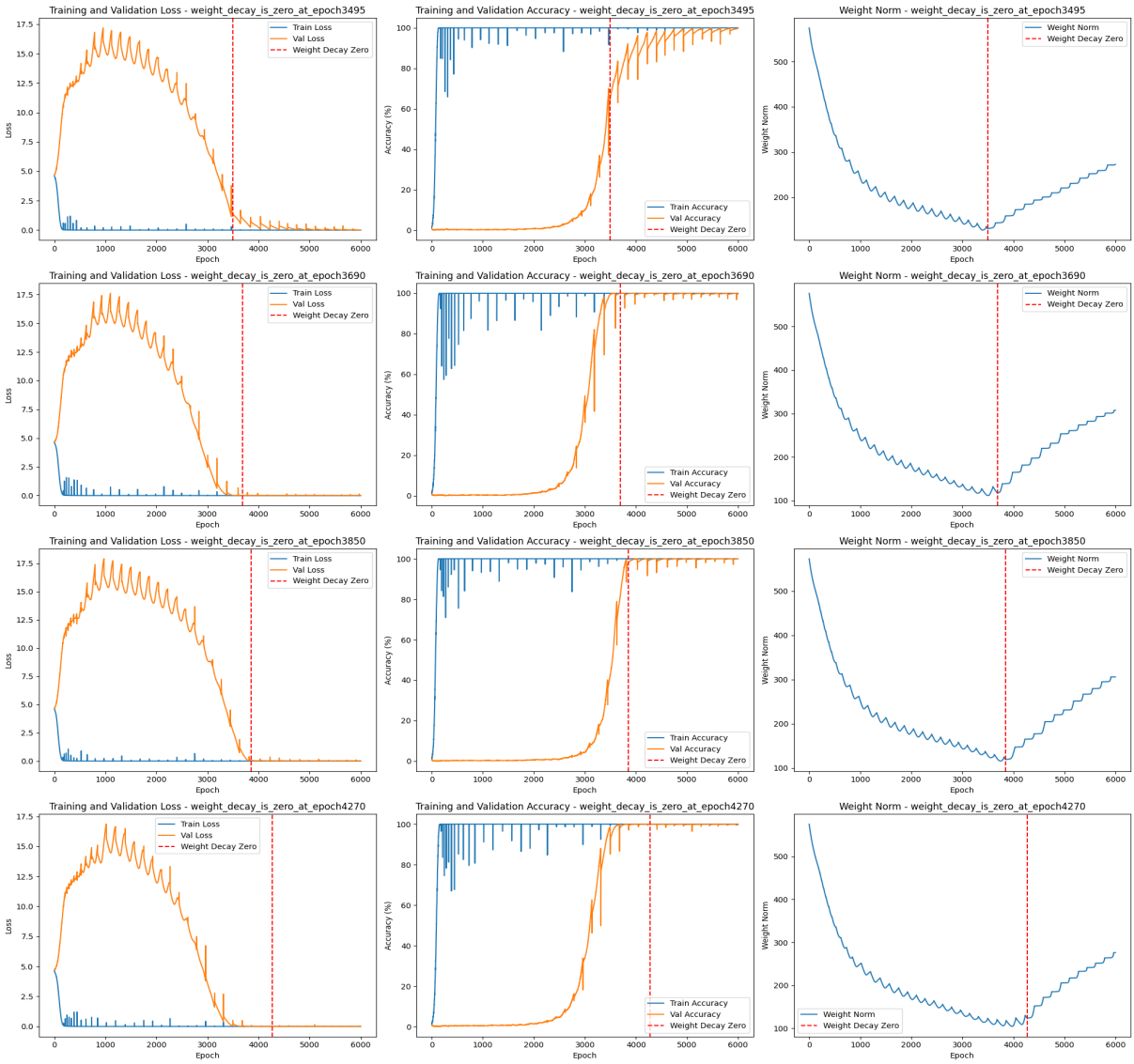


Figure 7.6: Training and validation loss, accuracy, and weight norm dynamics for the LSTM model trained on the same dataset. Weight decay is initially applied and then set to zero at higher epochs (3495, 3690, 3850, and 4270). Each row represents a different experiment with weight decay removed at a specific epoch. Each column provides insights into different metrics. The red dashed lines indicate the epochs where weight decay was removed.

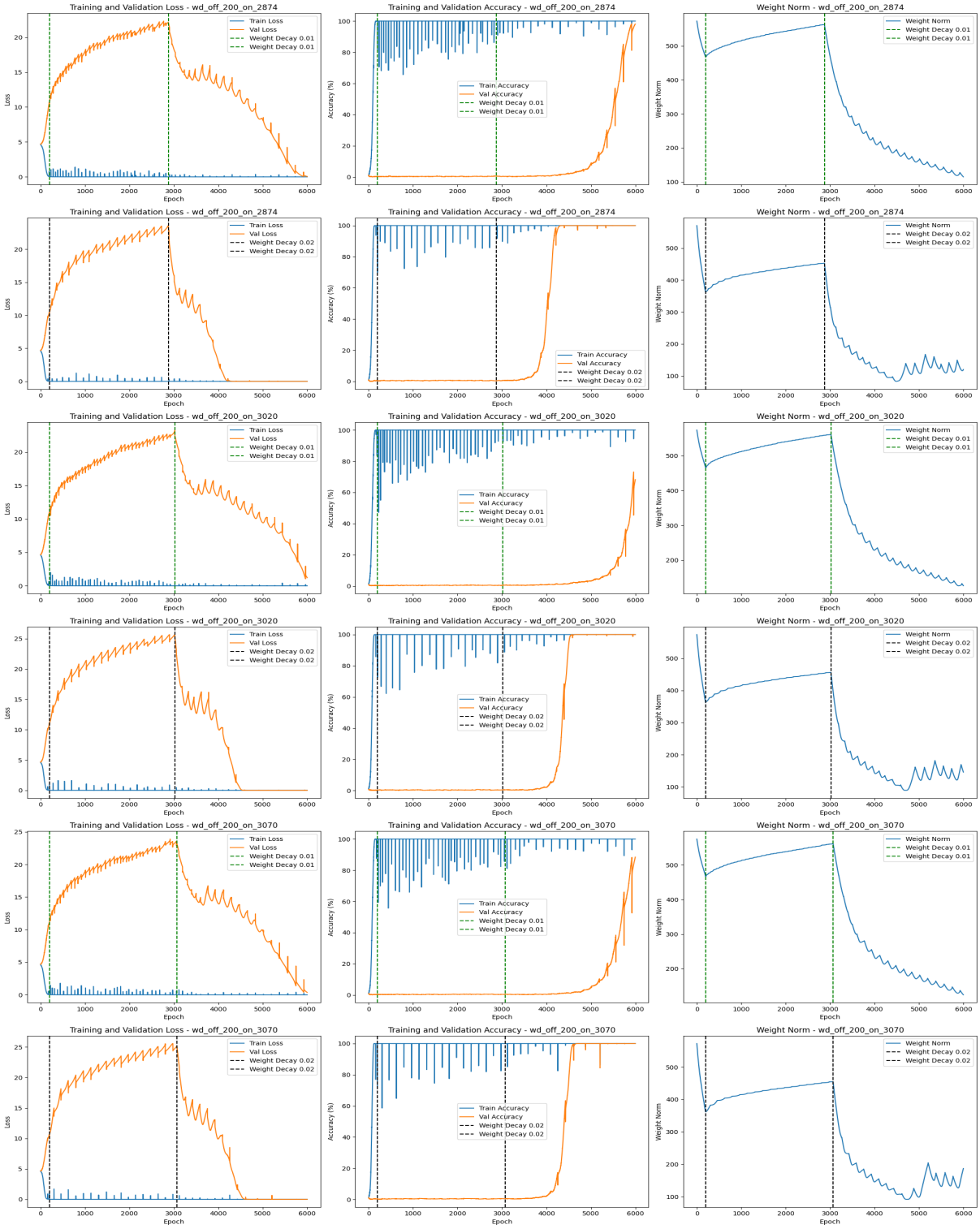
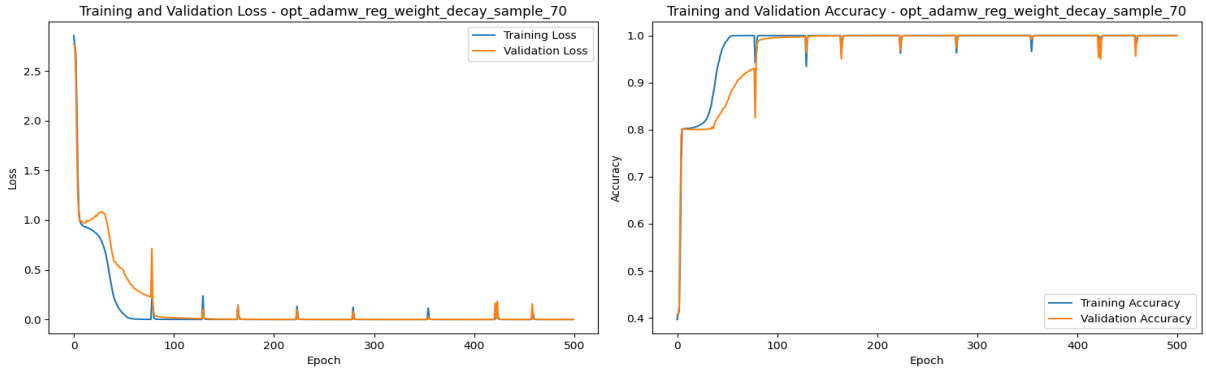
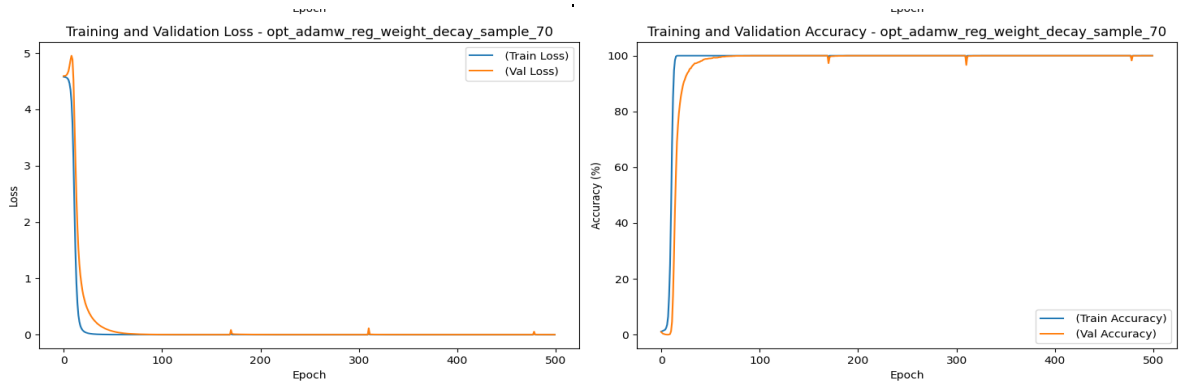


Figure 7.7: Training and validation loss, accuracy, and weight norm dynamics for the LSTM model trained on the same dataset with varying weight decay scenarios. Each row represents a different experimental setting where weight decay was dynamically adjusted. The black and green dashed lines indicate the points where weight decay of 0.01 and 0.02, respectively, were applied or removed.

7.3 Weight Initialization Method



(a) RNN Model



(b) LSTM Model

Figure 7.8: Training and validation metrics for (a) RNN and (b) LSTM models with AdamW optimizer at 70% sample size. Weight initialization was scaled up using a method from Liu et al. [17].