

# Pattern Recognition

## Fractional-order Jacobian Matrix Differentiation and Its Application in Artificial Neural Networks --Manuscript Draft--

<b>Manuscript Number:</b>	PR-D-25-03298
<b>Article Type:</b>	Full Length Article
<b>Section/Category:</b>	Innovative Applications of Pattern Recognition
<b>Keywords:</b>	Fractional-order optimization methods; Fractional-order matrix differentiation; Fractional-order Jacobian matrix; Fractional-order gradient descent method; Artificial neural networks
<b>Abstract:</b>	<p>Fractional-order differentiation has many characteristics different from integer-order differentiation. These characteristics can be applied to the optimization algorithms of artificial neural networks to obtain better results. However, due to insufficient theoretical research, at present, there is no fractional-order matrix differentiation method that is perfectly compatible with automatic differentiation (Autograd) technology. Therefore, we propose a fractional-order matrix differentiation calculation method. This method is introduced by the definition of the integer-order Jacobian matrix. We denote it as fractional-order Jacobian matrix differentiation. Through fractional-order Jacobian matrix differentiation, we can carry out the matrix-based fractional-order chain rule. Based on the Linear module and the fractional-order differentiation, we design the fractional-order Autograd technology to enable the use of fractional-order differentiation in hidden layers, thereby enhancing the practicality of fractional-order differentiation in deep learning. In the experiment, according to the PyTorch framework, we design fractional-order Linear (FLinear) and replace nn.Linear in the multilayer perceptron with FLinear. Through the qualitative analysis of the training set and validation set Loss, the quantitative analysis of the test set indicators, and the analysis of time consumption and GPU memory usage during model training, we verify the superior performance of fractional-order Jacobian matrix differentiation and prove that it is an excellent fractional-order gradient descent method in the field of deep learning.</p>

Xiaojun Zhou

School of Information Science and Engineering, Yunnan University, Kunming City,  
China, 650091

[zxjssldqm@163.com](mailto:zxjssldqm@163.com) (Xiaojun Zhou) and [zhaochunna@ynu.edu.cn](mailto:zhaochunna@ynu.edu.cn) <sup>\*</sup>(Chunna Zhao)

April 30, 2025

Submission of Manuscript: "Fractional-order Jacobian Matrix Differentiation and Its  
Application in Artificial Neural Networks"

Dear Editor,

I am pleased to submit our research paper, "Fractional-order Jacobian Matrix  
Differentiation and Its Application in Artificial Neural Networks", for consideration in  
Pattern Recognition.

This work introduces a novel fractional-order matrix differentiation method, derived  
from integer-order Jacobian differentiation, which we name **fractional-order Jacobian  
matrix differentiation ( $\mathbf{J}^\alpha$ )**. Leveraging  $\mathbf{J}^\alpha$ , we devise fractional-order automatic  
differentiation (Autograd) techniques, enabling efficient computation of fractional-  
order gradient descent (FGD) and its variants in artificial neural networks (ANNs). A  
key advancement is our method's ability to directly manipulate computational graph  
nodes, resolving the longstanding challenge of FGD's application to ANN hidden layers.

Our solution bridges this gap by proposing a method compatible with Autograd,  
marking a significant contribution to both fractional calculus and neural network  
optimization.

We believe this work's novelty, technical rigor, and relevance to Pattern Recognition's scope warrant its inclusion in your journal.

Thank you for considering our manuscript. We look forward to the opportunity to contribute to your esteemed publication.

Sincerely,

Xiaojun Zhou

## Highlights

### **Fractional-order Jacobian Matrix Differentiation and Its Application in Artificial Neural Networks\***

Xiaojun Zhou, Chunna Zhao, Yaqun Huang, Chengli Zhou, Junjie Ye, Kemeng Xiang

- Proposed fractional-order Jacobian matrix differentiation ( $\mathbf{J}^\alpha$ ) based on integer-order differentiation, revealing its block matrix structure and suitability for ANN applications.
- Implicit fractional-order matrix differentiation enhances FGD computational efficiency by avoiding explicit matrix element operations.
- Explores FGD in ANN hidden layers, integrates with Autograd, and proposes fractional-order Autograd.
- Re-defines FGD application rules in ANNs, eliminating the need for convergence proofs.

# Fractional-order Jacobian Matrix Differentiation and Its Application in Artificial Neural Networks<sup>★</sup>

Xiaojun Zhou<sup>a,b</sup>, Chunna Zhao<sup>a,1</sup>, Yaqun Huang<sup>a</sup>, Chengli Zhou<sup>a</sup>, Junjie Ye<sup>a</sup>, Kemeng Xiang<sup>a</sup>

<sup>a</sup>*School of Information Science and Engineering, Yunnan University, Kunming, 650500, China*

<sup>b</sup>*Lijiang Normal University, Lijiang, 674199, China*

---

## Abstract

Fractional-order differentiation has many characteristics different from integer-order differentiation. These characteristics can be applied to the optimization algorithms of artificial neural networks to obtain better results. However, due to insufficient theoretical research, at present, there is no fractional-order matrix differentiation method that is perfectly compatible with automatic differentiation (Autograd) technology. Therefore, we propose a fractional-order matrix differentiation calculation method. This method is introduced by the definition of the integer-order Jacobian matrix. We denote it as fractional-order Jacobian matrix differentiation ( $\mathbf{J}^\alpha$ ). Through  $\mathbf{J}^\alpha$ , we can carry out the matrix-based fractional-order chain rule. Based on the Linear module and the fractional-order differentiation, we design the fractional-order Autograd technology to enable the use of fractional-order differentiation in hidden layers, thereby enhancing the practicality of fractional-order differentiation in deep learning. In the experiment, according to the PyTorch framework, we design fractional-order Linear (FLinear) and replace `nn.Linear` in the multilayer perceptron with FLinear. Through the qualitative analysis of the training set and validation set *Loss*, the quantitative analysis of the test set indicators, and the analysis of time consumption and GPU memory usage during model training, we verify the superior performance of  $\mathbf{J}^\alpha$  and prove that it is an excel-

---

<sup>★</sup>This research was funded by the Yunnan University Graduate Research Innovation Fund Project.

*Email addresses:* zxjssldqm@163.com (Xiaojun Zhou), zhaochunna@ynu.edu.cn (Chunna Zhao), huangyq@ynu.edu.cn (Yaqun Huang), chenglizhou@mail.ynu.edu.cn (Chengli Zhou), yejunjie\_cdx@163.com (Junjie Ye), kemengxiang6@gmail.com (Kemeng Xiang)

<sup>1</sup>Corresponding author

lent fractional-order gradient descent method in the field of deep learning. The data and code are available at Github.

*Keywords:*

Fractional-order optimization methods, Fractional-order matrix differentiation, Fractional-order Jacobian matrix, Fractional-order gradient descent method, Artificial neural networks

---

## 1. Introduction

Integer-order matrix differentiation is an important theoretical basis for gradient calculation in artificial neural networks (ANN). Based on it, scholars have proposed many high-performance deep learning optimizers, which have promoted the development of ANN technology. Deep learning optimization algorithms have always been improved from the perspectives of first-order and second-order differentiations. However, with the development of fractional-order differentiation, some scholars began to study deep learning optimization algorithms from the perspective of fractional-order differentiation. These methods can be collectively referred to as fractional-order gradient descent (FGD) methods.

FGD methods have different characteristics from integer-order methods in theory. These characteristics include having built-in advanced learning rate scheduling strategies and being less likely to get stuck in saddle points (see Appendix A), as well as having built-in regularization capabilities (proof in Appendix B). Due to these advantages, scholars have attempted to introduce fractional calculus into deep learning and have achieved promising results [1, 2, 3]. However, in ANNs, data are represented in tensor form. Consequently, when fractional differentiation is extended from scalars to tensors, the fractional matrix differentiation within the hidden layers of ANNs cannot be computed in the same manner as integer-order differentiation, making the matrix-form chain rule inapplicable. As a result, fractional-order differentiation exhibits poor compatibility with integer-order automatic differentiation (Autograd) frameworks.

Currently, there are three approaches to applying FGD methods in ANNs: (1) Start from the perspective of symbolic differentiation to study FGD methods. To avoid cal-

culating fractional-order matrix differentiation and the subsequent chain calculations in ANNs, they only use fractional-order matrix differentiation on the Mean Squared Error Loss function (MSELoss). That is, they use the fractional-order power function differentiation formula to solve the differentiation of child nodes in the root node, then perform back-propagation based on the chain rule, and participate in the calculation of gradients of intermediate and leaf nodes. In fact, the differentiations of intermediate and leaf nodes are obtained using integer-order Autograd technology. No matter how the MSELoss is transformed, essentially, these are methods that can only use fractional-order differentiation at the output layer [4, 5, 6, 7, 8]. (2) Transform the Grünwald-Letnikov (G-L) formula and use Autograd to make the gradient solution become the nonlinear accumulation of gradients at different time nodes. Essentially, they are integer-order optimization methods with special momentum and special learning rates [9, 10, 11]. (3) Abandon the chain rule and use a specific symbolic differentiation method to solve gradients in specific ANNs based on Leibniz's rule [12].

The above methods conclude that fractional-order differentiation is superior to integer-order differentiation through theoretical analysis and experimental comparison. However, in practical application scenarios, the defects of these methods are also obvious. On the one hand, scholars have ignored the combination of relevant methods with mainstream deep learning frameworks and have not designed fractional-order deep learning optimizers from the perspective of Autograd technology. This may affect their further promotion in the field of deep learning optimization methods. On the other hand, many fractional-order parameter matrix gradients are not derived from Jacobian matrix differentiation like integer-order matrices. Many of them are computationally complex during the solution process, and when the formulas are transformed, the basis of fractional-order matrix differentiation is weakened to varying degrees [13]. In short, the fractional-order deep learning optimizers proposed based on existing FGD methods are not sufficient to comprehensively surpass the numerous integer-order deep learning optimizers.

In addition, scholars have conducted convergence analysis on the proposed FGD methods, and most of them emphasize faster convergence rate on the training set than the corresponding integer-order methods. However, the existing FGD methods and

their variants are improvements based on existing first-order optimization algorithms, and their convergence performance is the same as that of the corresponding integer-order optimization algorithms.

Since there is a linear mapping on the gradient matrix between some existing FGD methods and integer-order optimization methods or their variants, adjusting their fractional-order orders can ultimately be regarded as adjusting the learning rate. This further affects the engineering application of FGD methods. Therefore, in order for FGD methods to have long-term development and application in deep learning, we need to re-examine FGD methods and redefine them according to the research ideas of Autograd technology and integer-order matrix differentiation. In this process, the biggest problem faced by FGD methods is that fractional-order differentiation cannot be applied to the hidden layers of ANNs [14, 15, 16, 17, 18, 19, 20].

Thus, This paper proposes fractional-order Jacobian matrix differentiation  $\mathbf{J}^\alpha$  based on the definition of integer-order Jacobian matrices and fractional-order symbolic differentiation. In the experiment, combined with the mainstream deep learning framework PyTorch, the Linear module is reconstructed into a fractional-order Linear module. In fact, this paper preliminarily proposes fractional-order Autograd technology. By similarly reconstructing other modules in torch.nn, we can also establish a complete ecosystem of fractional-order Autograd technology. The biggest difference between the work in this paper and previous FGD methods is that through calculating the differentiations of each node on the computation graph, the fractional-order gradient is finally obtained. Unlike existing FGD methods that make modifications in `Optimizer.step()`. In fact, this paper performs fractional-order gradient calculation in `Loss.backward()` (see Figure 1).

In specific details, we extract the fractional-order differentiation matrix required for solving the gradient matrix from  $\mathbf{J}^\alpha$ . Its size is the same as that of the integer-order differentiation matrix. In contrast, for integer-order cases, only multiple completely identical differentiation matrices can be defined from the definition of  $\mathbf{J}^\alpha$ . When  $\alpha = 1.0$ , the block matrix obtained is the same as the differentiation matrix obtained in traditional integer-order cases, that is, only one differentiation matrix can be obtained. On the contrary, fractional-order cases can define multiple different differentiation matri-



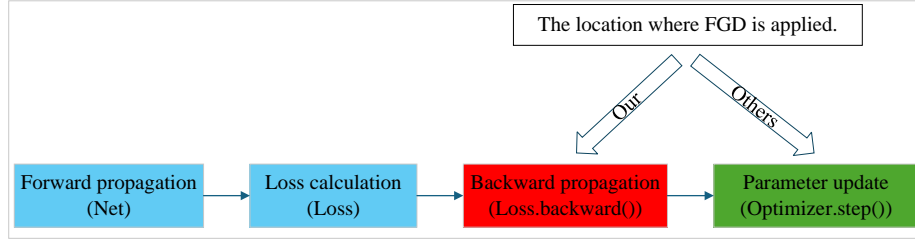


Figure 1: Location differences in operational scope between the proposed method and existing approaches in PyTorch.

ces. Since scalar-based calculations are impractical in ANNs, we need to convert the calculations of each differentiation matrix in  $\mathbf{J}^\alpha$  into matrix-based calculations. The size of the differentiation matrix defined by  $\mathbf{J}^\alpha$  is related to the column size of the parameter matrix (usually the weight matrix in this paper), and each parameter matrix has a different size. Therefore, each parameter matrix can select a different number of differentiation matrices for gradient calculation. In matrix-based algorithm design, to reduce the complexity of the problem, in this paper, we only select the first block matrix in  $\mathbf{J}^\alpha$  (i.e., the first block matrix on the diagonal) as the fractional-order differentiation matrix to participate in the gradient calculation of the weight matrix in the leaf node. Compared with previous research and applications, the fractional-order weight gradient matrix in this paper is not a linear mapping of the integer-order gradient matrix, nor is it equivalent to a variant of the gradient descent (GD) method. On the contrary, this paper’s method fully utilizes the learning rate scheduling strategy and regularization characteristics of fractional-order optimization.

In summary, in this paper: (1) We expound on the theoretical method of  $\mathbf{J}^\alpha$ . (2) We present a fractional-order linear function differentiation formula suitable for the field of deep learning. (3) In a given multilayer perceptron (MLP), following the idea of Autograd technology, we combine the fractional-order linear function differentiation formula with  $\mathbf{J}^\alpha$  and present a gradient solution algorithm based on matrix calculations. (4) We conduct an experiment on a regression task based on MLP. In the experiment, using the deep learning framework PyTorch and combining Autograd technol-

ogy, we reconstruct the Linear module, adding a fractional-order backward function and a fractional-order matrix differentiation solver in it. In fact, we design a fractional-order Linear module (FLinear) in PyTorch. And FLinear is a module that contains Linear. In the experiment, we replace the Linear layer in the ANN with the FLinear layer, enabling the model to use the matrix-based fractional-order chain rule during back-propagation. Through performance analysis of different orders (when  $\alpha = 1.0$ , FLinear is the same as Linear),  $\mathbf{J}^\alpha$  shows superiority.

In deep learning, the method proposed in this paper enables fractional-order matrix differentiation to be applied inside ANNs, not just at the output layer. This allows the FGD method to no longer rely on integer-order Autograd technology and makes it a method that includes integer-order differentiation. Therefore, the contributions of this paper can be summarized as follows:

- Based on integer-order Jacobian matrix differentiation,  $\mathbf{J}^\alpha$  is proposed, and its block matrix characteristics are studied. The characteristics of the first block matrix of the fractional-order and the integer-order are compared and analyzed experimentally.
- The proposed fractional-order matrix differentiation method can avoid explicit operations on matrix elements, thus improving the computational efficiency of the FGD method.
- This paper focuses on applying the FGD method in the hidden layers of ANNs and explains and experimentally analyzes the FGD method and its variants from a mathematical theory perspective. Closely integrated with Autograd technology, its application ability is generalized. This makes the FGD method no longer a method that only uses fractional-order differentiation at the root node.
- Since this paper studies the FGD method based on the idea of Autograd technology, in fact, we have found a breakthrough application for fractional-order automatic differentiation (FAutograd) technology. This makes the FGD method and its variants no longer a linear or simple non-linear transformation of the integer-order weight matrix gradient in `Optimizer.Step()`. Instead, it is an operation on

the computation graph in `Loss.backward()`.

- Through extensive research on the application of the FGD method in ANNs, this paper re-states and defines the application rules of the FGD method and clarifies the non-necessity of proving the convergence of the FGD method.

The remaining parts of this paper are organized as follows: Section 2 introduces deep learning optimization strategies and the current research status of the FGD method, as well as the application method of fractional-order differentiation in MLPs. Section 3 proposes the method in this paper, including  $\mathbf{J}^\alpha$ , the fractional-order differentiation formula in ANNs, and the calculation of the fractional-order gradient matrix. Section 4 is about the experiments. Section 5 is the summary.

## 2. Related Work and Motivation

In this section, we will introduce existing deep learning strategies and FGD methods, and expound on their relationship. On this basis, we will introduce in detail the application of FGD methods in ANNs. Finally, we will explain the motivation for the work in this paper.

### 2.1. Deep Learning Optimization Strategies and Fractional-order Gradient Descent

In the parameter optimization methods of neural networks, integer-order deep learning optimization algorithms are divided into first-order and second-order optimization methods. The FGD method discussed in this paper is an extension based on first-order optimization methods. Therefore, this section first reviews first-order optimization methods. Then, it explores the learning rate scheduling strategies in deep learning optimization methods. Finally, it re-examines the current research status of existing FGD methods and summarizes them.

**Integer-order Deep Learning Optimization Methods:** In the integer-order deep learning framework PyTorch, the Stochastic Gradient Descent (SGD) optimizer is an integrated solution. It includes the SGD method [21], the Momentum method [22], and the Nesterov Accelerated Gradient (NAG) method [23]. These three optimization methods are achieved by adjusting the parameters of the optimizer. Among them, the

SGD method is the most basic optimization method. Due to its inherent problems [24], scholars proposed the Momentum and NAG methods to improve the SGD optimizer. The Momentum method uses past gradient information, while the NAG method uses future gradient information to optimize the parameters being iterated. In practical application scenarios, the SGD optimizer is usually combined with them to accelerate model convergence and help the model escape local optima. Another major category of first-order optimization methods is adaptive learning rate optimizers. These optimizers include Adagrad [25], Adadelta [26], RMSprop [27], and Adam [28], etc. These adaptive learning rate optimizers use the information of the first-order or second-order moments of the gradient, or both, and achieve iterative updates of the parameter matrix through a non-linear transformation of the gradient matrix. They are equivalent to improving the step size of the gradient matrix in each iteration.

In fact, adaptive learning rate optimizers are equivalent to giving each element of the gradient matrix a separate learning rate, which is a non-linear transformation of the matrix.

**Learning Rate Scheduling Strategies:** In addition to using the first-order and second-order moments of the gradient, another important method in the training of ANNs is the learning rate scheduling strategy. During the iteration process, these strategies dynamically adjust the learning rate to increase the step size of each iteration. Such techniques include learning rate decay, cosine annealing [29], and warm-up [30]. In fact, they are scalar multiplications of the matrix, which are linear transformations of the matrix.

That is to say, there is an important difference between integer-order deep learning optimization methods and learning rate scheduling strategies at the matrix level. In terms of the transformation of the gradient matrix, integer-order deep learning optimization methods are non-linear transformations. On the contrary, learning rate adjustment strategies are linear transformations. However, from the longitudinal process of iteration, most learning rate decay methods, cosine annealing, and warm-up can be regarded as non-linear transformations on the gradient matrix. Based on these discussions, when introducing existing FGD methods, we can find their inherent characteristics and determine whether the gradient matrices are linear transformations or

non-linear transformations in `Optimizer.step()`.

**Existing FGD Methods:** In the research of ANNs, FGD methods use fractional-order differentiation and integer-order optimization techniques to develop new deep learning optimization techniques. Some of these methods use integer-order gradients for solving. References [9, 10, 11] are examples. By transforming the G-L formula and combining it with automatic differentiation technology, they finally derive the fractional-order gradient. Essentially, they can be classified as a learning rate scheduling strategy or a momentum method with a sliding window. In fact, they can also be regarded as a special integer-order momentum method. Moreover, they are based on numerical differentiation and are not convenient for compatibility with Autograd technology. Similarly, references [31, 32] also use numerical differentiation. Although the Caputo definition is used and fractional-order differentiation is specifically applied to `MSELoss` at the root node, from the perspective of ANNs, this is equivalent to using fractional-order differentiation at the output layer while maintaining integer-order differentiation in the hidden layers. Another category of FGD methods is based on symbolic differentiation for solving. In particular, the fractional-order differentiation formula of quadratic functions is used for solving, that is, fractional-order differentiation is applied to `MSELoss` [5]. Reference [4] transforms the quadratic function before applying fractional-order differentiation to alleviate the gradient explosion problem, it caused by the loss of gradient direction due to the absolute value symbol. Reference [12] also uses symbolic differentiation, but it abandons the chain rule and uses Leibniz’s rule in specific ANNs. In fact, by transforming the differentiation formula, the use of absolute values is avoided. However, including reference [7], these FGD methods using symbolic differentiation may have a linear mapping with integer-order optimization methods. This linear mapping can be achieved by adjusting the learning rate in integer-order deep learning optimizers. Even if the gradient matrix of some of these methods is no longer a linear mapping relationship after formula transformation, essentially, they operate on the basis of the integer-order gradient matrix, rather than directly solving the fractional-order gradient matrix of the leaf nodes on the computation graph. The advantage of FGD methods based on symbolic differentiation is that they can use the idea of Autograd technology. However, generally speaking, these methods may

have two problems: (1) The fractional-order differentiation formula is heavily modified to avoid complex fractional-order matrix differentiation. (2) When performing the chain rule in matrix form, to solve row-column mismatch between two differentiation matrices in matrix multiplication, matrix multiplication is changed to the Hadamard product.

## 2.2. Four Cases of Solving Weights with Fractional-order Differentiation in Multilayer Perceptrons

To further elaborate on the current research status of the FGD method in ANNs, we use a MLP with two linear functions and a quadratic function as the ANN, and introduce four cases of using the FGD method. This ANN can be expressed as Eq. (1), and the corresponding computation graph is shown in Figure 2. Specifically, to simplify the description of the example problem and meet the requirements of hyperparameters for subsequent experiments, we first assume that the variables and parameters in Eq. (1) are all scalars, and there is no activation function after each linear function.

$$\begin{cases} y_1 = xw_1 + b_1 \\ y_2 = y_1w_2 + b_2 \\ L = (y_2 - y_{\text{label}})^2 \end{cases} \quad (1)$$

where  $x$  represents the input data,  $\{w_1, w_2\}$  are the weights,  $\{b_1, b_2\}$  are the biases,  $y_{\text{label}}$  is the label, and  $L$  is the value of the loss function (*Loss*).

In Figure 2, through the chain rule and fractional-order differentiation, four methods for calculating the gradients of leaf nodes can be summarized and defined. Taking  $w_1$  as an example, the definitions of its four gradient calculation methods are as follows:

Case 1:

$$\nabla_{w_1}^{(1,1,\alpha)} L(w_1; x, b_1, y_1, y_2) := \frac{\partial L}{\partial y_2} \frac{\partial y_2}{\partial y_1} \frac{\partial^\alpha y_1}{\partial (w_1)^\alpha} \quad (2)$$

Here,  $\nabla_{w_1}^{(1,\alpha,\alpha)} L(w_1; x, b_1, y_1, y_2)$  means that both integer-order and fractional-order are used when calculating the gradient of  $w_1$ .  $\frac{\partial^\alpha y_1}{\partial (w_1)^\alpha}$  represents the  $\alpha$ -th order partial derivative of  $y_1$  with respect to  $w_1$ .

Case 2:

$$\nabla_{w_1}^{(1,\alpha,\alpha)} L(w_1; x, b_1, y_1, y_2) := \frac{\partial L}{\partial y_2} \frac{\partial^\alpha y_2}{\partial (y_1)^\alpha} \frac{\partial^\alpha y_1}{\partial (w_1)^\alpha} \quad (3)$$

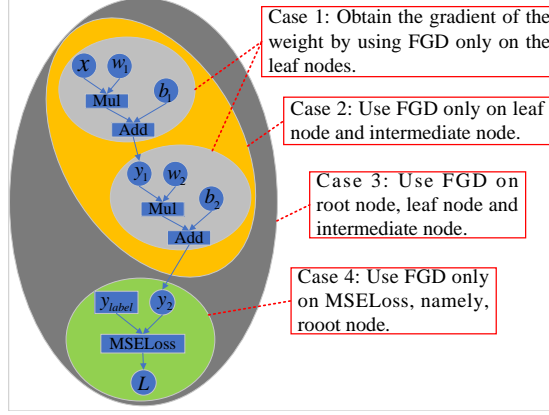


Figure 2: Computational graph of ANN and 4 cases of FGD in back-propagation.

Case 3:

$$\nabla_{w_1}^\alpha L(w_1; x, b_1, y_1, y_2) := \frac{\partial^\alpha L}{\partial (y_2)^\alpha} \frac{\partial^\alpha y_2}{\partial (y_1)^\alpha} \frac{\partial^\alpha y_1}{\partial (w_1)^\alpha} \quad (4)$$

Here,  $\nabla_{w_1}^\alpha L(w_1; x, b_1, y_1, y_2)$  represents the fractional-order gradient of  $L$  with respect to  $w_1$ .

Case 4:

$$\nabla_{w_1}^{(\alpha, 1, 1)} L(w_1; x, b_1, y_1, y_2) := \frac{\partial^\alpha L}{\partial (y_2)^\alpha} \frac{\partial y_2}{\partial y_1} \frac{\partial y_1}{\partial w_1} \quad (5)$$

Among the four gradient calculation formulas, Case 4 corresponds to some characteristics of existing FGD methods [4, 5, 6, 7, 8]. In back-propagation, Case 4 is a method used only at the output layer. Conversely, Cases 1, 2, and 3 all involve the problem of fractional-order matrix differentiation in the internal calculation of neural networks. That is, according to the application of existing FGD methods in Sections 1 and 2.1, in the hidden layers of ANNs, Case 4 is a method that must rely on integer-order differentiation to obtain parameter gradients. It is not independent of integer order, and its so-called fractional order does not truly incorporate integer order.

### 2.3. Motivation

Why can't existing FGD methods be applied to the hidden layers of ANNs? In many studies on fractional-order differentiation, variables and parameters are assumed

to be scalars. However, in actual engineering applications, the data, weights, and biases being trained are tensors [13, 33, 34, 35]. In back-propagation, the multiplication between tensors must follow the rules of matrix multiplication. That is, when two differentiation matrices are multiplied, their row and column dimensions must be equal. In integer-order matrix differentiation, as long as the matrices in forward propagation satisfy the operation rules of matrix multiplication, the differentiation matrices obtained through the integer-order matrix differentiation solution rules in back-propagation also satisfy the matrix multiplication rules. But when we introduce the integer-order calculation idea (such as the Autograd technology of the deep learning framework PyTorch) into fractional-order and use fractional-order differentiation to solve the gradients of matrix functions, we cannot obtain a differentiation matrix that satisfies the matrix multiplication rules. In deep learning optimizers, this differentiation matrix needs to participate in chain-based solutions to obtain the gradient matrix. Therefore, we need to study a kind of fractional-order matrix differentiation method so that the obtained differentiation matrix satisfies the matrix-based chain rule.

### 3. Method

This section systematically introduces  $\mathbf{J}^\alpha$ . Meanwhile, based on previous studies, a fractional-order differentiation formula suitable for the Autograd technology approach is presented. On this basis, the implicit calculation method of the fractional-order gradient matrix is expounded.

#### 3.1. Fractional-order Jacobian Matrix Differentiation

Based on the discussion of the motivation in Section 2.3, when data and parameters change from scalars to tensors during back-propagation, the defects of existing FGD methods can be known. To solve this problem, we need to make the sizes of the fractional and integer-order differentiation matrices the same. At the same time, when the fractional order  $\alpha = 1.0$ , the obtained differentiation matrix should be equal to that obtained by the integer-order method. For this purpose, based on the definition of the integer-order Jacobian matrix, we use fractional-order symbolic differentiation on the



matrix function to obtain the fractional-order Jacobian matrix. Finally, a block matrix that satisfies the matrix multiplication rules during back-propagation is extracted from the fractional-order Jacobian matrix. The details are as follows:

**Definition 1.** Let the matrix function  $\mathbf{F}(\mathbf{X}) \in \mathbb{R}^{p \times q}$ , and the matrix  $\mathbf{X} \in \mathbb{R}^{m \times n}$ . Their vectorizations are defined as

$$\text{vec}(\mathbf{F}(\mathbf{X})) := [f_{11}(\mathbf{X}), \dots, f_{p1}(\mathbf{X}), \dots, f_{1q}(\mathbf{X}), \dots, f_{pq}(\mathbf{X})]^T \in \mathbb{R}^{pq} \quad (6)$$

$$\text{vec}(\mathbf{X}) := [x_{11}, \dots, x_{m1}, \dots, x_{1n}, \dots, x_{mn}]^T \in \mathbb{R}^{mn} \quad (7)$$

Then the fractional-order Jacobian matrix of the matrix function  $\mathbf{F}(\mathbf{X})$  is defined as

$$\mathbf{J}^\alpha = \nabla_{(\text{vec} \mathbf{X})}^\alpha \mathbf{F}(\mathbf{X}) := \frac{\partial^\alpha \text{vec} \mathbf{F}(\mathbf{X})}{\partial ((\text{vec} \mathbf{X})^T)^\alpha} \in \mathbb{R}^{pq \times mn} \quad (8)$$

Its specific representation is Eq. (9).

$$\mathbf{J}^\alpha = \begin{bmatrix} \frac{\partial^\alpha f_{11}}{\partial ((\text{vec} \mathbf{X})^T)^\alpha} \\ \vdots \\ \frac{\partial^\alpha f_{p1}}{\partial ((\text{vec} \mathbf{X})^T)^\alpha} \\ \vdots \\ \frac{\partial^\alpha f_{1q}}{\partial ((\text{vec} \mathbf{X})^T)^\alpha} \\ \vdots \\ \frac{\partial^\alpha f_{pq}}{\partial ((\text{vec} \mathbf{X})^T)^\alpha} \end{bmatrix} = \begin{bmatrix} \frac{\partial^\alpha f_{11}}{\partial (x_{11})^\alpha} & \dots & \frac{\partial^\alpha f_{11}}{\partial (x_{m1})^\alpha} & \dots & \frac{\partial^\alpha f_{11}}{\partial (x_{1n})^\alpha} & \dots & \frac{\partial^\alpha f_{11}}{\partial (x_{mn})^\alpha} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \frac{\partial^\alpha f_{p1}}{\partial (x_{11})^\alpha} & \dots & \frac{\partial^\alpha f_{p1}}{\partial (x_{m1})^\alpha} & \dots & \frac{\partial^\alpha f_{p1}}{\partial (x_{1n})^\alpha} & \dots & \frac{\partial^\alpha f_{p1}}{\partial (x_{mn})^\alpha} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \frac{\partial^\alpha f_{1q}}{\partial (x_{11})^\alpha} & \dots & \frac{\partial^\alpha f_{1q}}{\partial (x_{m1})^\alpha} & \dots & \frac{\partial^\alpha f_{1q}}{\partial (x_{1n})^\alpha} & \dots & \frac{\partial^\alpha f_{1q}}{\partial (x_{mn})^\alpha} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \frac{\partial^\alpha f_{pq}}{\partial (x_{11})^\alpha} & \dots & \frac{\partial^\alpha f_{pq}}{\partial (x_{m1})^\alpha} & \dots & \frac{\partial^\alpha f_{pq}}{\partial (x_{1n})^\alpha} & \dots & \frac{\partial^\alpha f_{pq}}{\partial (x_{mn})^\alpha} \end{bmatrix} \quad (9)$$

In Eq. (9), when  $\alpha = 1.0$ ,  $\mathbf{J}^\alpha = \mathbf{J}$ , which is the integer-order Jacobian matrix.

Now, let the matrix function  $\mathbf{F}(\mathbf{X}) = \mathbf{A}^{p \times m} \mathbf{X}^{m \times n} \in \mathbb{R}^{p \times n}$  in Definition 1, and the coefficient matrix  $\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{p1} & \dots & a_{pm} \end{bmatrix}$ . According to Eq. (9), for integer-order matrix differentiation, we have the following formula

$$\mathbf{J} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{O} & \cdots & \mathbf{O} \\ \mathbf{O} & \mathbf{A}_2 & \cdots & \mathbf{O} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{O} & \mathbf{O} & \cdots & \mathbf{A}_n \end{bmatrix} \quad (10)$$

where  $\mathbf{O}$  is a  $p \times m$  zero matrix, and  $\mathbf{A}_1 = \mathbf{A}_2 = \cdots = \mathbf{A}_n = \mathbf{A}$ . And  $\mathbf{A}^T \in \mathbb{R}^{m \times p}$ , which is the integer-order differentiation matrix. When we regard  $\mathbf{F}(\mathbf{X}) = \mathbf{A}^{p \times m} \mathbf{X}^{m \times n}$  as a linear layer of an ANN without bias,  $\mathbf{A}^T$  can be used as an intermediate node on the computation graph and participate in the calculation of the gradient matrices of each node on the computation graph.

Similarly, the calculation formula for fractional-order matrix differentiation is as follows

$$\mathbf{J}^\alpha = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1n} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2n} \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_{n1} & \mathbf{A}_{n2} & \cdots & \mathbf{A}_{nn} \end{bmatrix} \quad (11)$$

where  $\{\mathbf{A}_{11}, \cdots, \mathbf{A}_{1n}, \cdots, \mathbf{A}_{n1}, \cdots, \mathbf{A}_{nn}\} \in \mathbb{R}^{p \times m}$ .

Since the derivative of a constant is not zero in fractional-order differentiation, in Eq. (11), the block matrices on the non-diagonal are not zero matrices, and the  $n$  block matrices on the diagonal are also different. That is,  $\mathbf{J}^\alpha$  has  $n^2$  differentiation matrices, and each of these block matrices can be used as a differentiation matrix to participate in the chain-based derivative of fractional-order matrix differentiation. In particular, when  $\alpha = 1.0$  or  $n = 1$ ,  $\mathbf{J}^\alpha$  has only one differentiation matrix, which is the integer-order differentiation matrix.

### 3.2. Fractional-order Differentiation Formulas in Neural Networks

In many methods that apply the FGD method to ANNs, due to problems such as the upper and lower bounds of fractional-order differentiation, step size, polynomial calculation, and integration, they usually transform the formulas of GL, Riemann-Liouville (RL), or Caputo to meet the efficiency requirements in engineering applications.

However, in this paper, based on the technical idea of Autograd, we want to study the fractional-order Autograd technology and use fractional-order differentiation in the neural network inside the ANN, that is, in the computation graph. Therefore, we apply fractional-order symbolic differentiation to fractional-order matrix differentiation instead of using numerical differentiations such as GL, RL, and Caputo.

Based on the discussions in previous sections and the following two reasons, we use linear functions as the discussion example.

(1) Although neural network models have become very complex and many classic models have emerged, the Linear layer based on linear functions is still an important module in many ANNs.

(2) In the experimental part of this paper, a time-series prediction task based on MLP is carried out, and its network model is composed of many Linear layers.

So, based on linear functions, we give a definition of the fractional-order differentiation formula proposed in this paper [36], as follows:

**Definition 2.** Let  $y = xw + b$ , then

$$\frac{\partial^\alpha y}{\partial w^\alpha} := {}^R_0\mathcal{D}_w^\alpha y = \frac{x}{\Gamma(2-\alpha)}|w|^{1-\alpha} + \text{sign}(w)\frac{b}{\Gamma(1-\alpha)}|w|^{-\alpha} \quad (12)$$

where the order  $\alpha \in (0, 2)$ . In this paper, to simplify the problem discussion and focus on first-order and fractional-order methods, we let  $\alpha \in (0, 1]$ .  $\text{sign}(\bullet)$  is the sign function, which aims to ensure that the gradient direction of the corresponding term is not lost. In deep learning,  $x$  is the input data,  $w$  is the weight, and  $b$  is the bias.

Since the chain rule in Autograd technology is based on symbolic differentiation, Eq. (12) as a symbolic differentiation formula meets the requirements of Autograd technology for differentiation formulas. At the same time, it can be noted that in Formula 12, since the study of the lower bound is not the focus of this paper, its lower bound is set to 0. This is a default lower bound. In fact, the differentiation formula of Definition 2 is obtained by statistical induction, that is, the lower bound is 0. This is consistent with the power-function differentiation formula obtained after using RL derivation. In engineering applications, different initialization methods are selected for parameters according to different models. Therefore, it should be noted that the lower bound of fractional-order differentiation here is different from the initial parameter values. At

the same time, a  $\text{sign}(\bullet)$  is added before the second term in Eq. (12). This is the same as most methods that apply fractional-order differentiation in engineering, which is to prevent complex numbers from being obtained when solving fractional powers. As a whole, this paper will also combine the gradient direction problem and explain the reason for adding the sign function in the second term from a new perspective.

Taking the ANN shown in Figure 2 as an example, it is composed of multiple Linear layers and an MSELoss. From the scalar perspective, in integer order, the gradient direction of any leaf node is not affected by the positive or negative value of the node in the Linear layer when calculating the gradient. The gradient direction is only affected by the MSELoss, that is, the influence of the back-propagated value when the root node is differentiated. In fact, if we ignore the influence of intermediate nodes on the final gradient value, we can think that in integer order, the gradient direction is determined by the loss function. This is determined by the convex or non-convex nature of the function. In integer order, usually, we only encounter the problems of differentiating first-power and second-power functions. However, in fractional order, since the symbolic differentiation is a linear operator, there is also zero-power differentiation. Zero-power is also an even-power. So, to ensure the unity of fractional-order Autograd and integer-order Autograd, we add a sign function before zero-power differentiation. In contrast, in linear functions, no sign function is added for first-power fractional-order differentiation.

Now, let the matrix function  $\mathbf{F}(\mathbf{X}) = \mathbf{A}^{p \times m} \mathbf{X}^{m \times n} + \mathbf{b}^{1 \times n}$  (In PyTorch, the bias is automatically expanded along rows to  $\mathbf{b}^{p \times n}$  due to broadcasting), then according to Formulas (9) and (11), we can obtain any fractional-order differentiation matrix in  $\frac{\partial \mathbf{F}(\mathbf{X})}{\partial \mathbf{X}}$ . Taking the calculation of the first differentiation matrix  $\mathbf{A}_{11}$  in Eq. (11) as an example, it can be expressed as Eq. (13).

$$\mathbf{A}_{11} = \begin{bmatrix} \frac{\partial^\alpha f_{11}}{\partial (x_{11})^\alpha} & \cdots & \frac{\partial^\alpha f_{11}}{\partial (x_{m1})^\alpha} \\ \vdots & & \vdots \\ \frac{\partial^\alpha f_{p1}}{\partial (x_{11})^\alpha} & \cdots & \frac{\partial^\alpha f_{p1}}{\partial (x_{m1})^\alpha} \end{bmatrix} \quad (13)$$

It can be known that in Eq. (14),

$$\begin{aligned}
f_{11} &= a_{11}x_{11} + a_{12}x_{21} + \cdots + a_{1m}x_{m1} + b_1 \\
f_{21} &= a_{21}x_{11} + a_{22}x_{21} + \cdots + a_{2m}x_{m1} + b_1 \\
&\vdots \\
f_{p1} &= a_{p1}x_{11} + a_{p2}x_{21} + \cdots + a_{pm}x_{m1} + b_1
\end{aligned} \tag{14}$$

According to Eq. (12), for  $\left\{ \frac{\partial^\alpha f_{11}}{\partial(x_{11})^\alpha}, \dots, \frac{\partial^\alpha f_{p1}}{\partial(x_{11})^\alpha}, \dots, \frac{\partial^\alpha f_{11}}{\partial(x_{m1})^\alpha}, \dots, \frac{\partial^\alpha f_{p1}}{\partial(x_{m1})^\alpha} \right\}$ , we have Eq. (15).

$$\left\{ \begin{aligned} \frac{\partial^\alpha f_{11}}{\partial(x_{11})^\alpha} &= \frac{a_{11}}{\Gamma(2-\alpha)} |x_{11}|^{1-\alpha} + \text{sign}(x_{11}) \frac{\sum_{i=1}^m a_{1i}x_{i1} - a_{11}x_{11} + b_1}{\Gamma(1-\alpha)} |x_{11}|^{-\alpha} \\ &\vdots \\ \frac{\partial^\alpha f_{p1}}{\partial(x_{11})^\alpha} &= \frac{a_{p1}}{\Gamma(2-\alpha)} |x_{11}|^{1-\alpha} + \text{sign}(x_{11}) \frac{\sum_{i=1}^m a_{pi}x_{i1} - a_{p1}x_{11} + b_1}{\Gamma(1-\alpha)} |x_{11}|^{-\alpha} \\ &\vdots \\ \frac{\partial^\alpha f_{11}}{\partial(x_{m1})^\alpha} &= \frac{a_{11}}{\Gamma(2-\alpha)} |x_{m1}|^{1-\alpha} + \text{sign}(x_{m1}) \frac{\sum_{i=1}^m a_{1i}x_{i1} - a_{11}x_{m1} + b_1}{\Gamma(1-\alpha)} |x_{m1}|^{-\alpha} \\ &\vdots \\ \frac{\partial^\alpha f_{p1}}{\partial(x_{m1})^\alpha} &= \frac{a_{p1}}{\Gamma(2-\alpha)} |x_{m1}|^{1-\alpha} + \text{sign}(x_{m1}) \frac{\sum_{i=1}^m a_{pi}x_{i1} - a_{p1}x_{m1} + b_1}{\Gamma(1-\alpha)} |x_{m1}|^{-\alpha} \end{aligned} \right. \tag{15}$$

Eq. (15) is the explicit calculation of fractional-order matrix differentiation. In computers, by converting the explicit calculation of matrix differentiation into implicit calculation, we can efficiently obtain the corresponding node's differentiation matrix and store it in the corresponding computation graph, and finally provide it to each leaf node to calculate the corresponding gradient matrix.

### 3.3. Calculation of Fractional-Order Gradient Matrix

The core problem of this paper is to use  $\mathbf{J}^\alpha$  to realize the application of the FGD method in hidden layers. As described in Section 2.2, Case 1, 2, and 3 are essentially the same type of problem. Therefore, to simplify the problem, the algorithms and experiments in this paper only take Case 1 as an example. And the ANN in the experimental part is also based on the network model of Eq. (1) to conduct qualitative and quantitative analyses on real-world time-series datasets.

As can be seen from Eq. (15), in Section 2.2, fractional-order matrix differentiation is obtained through explicit calculation. However, in application scenarios, this is very time-consuming. Therefore, in this section, we first need to convert the fractional-order matrix differentiation in Section 2.2 into implicit calculation, that is, all calculations are based on matrices.

According to Eq. (1) and Figure 2, in the neural network model example in this paper, the tensor form of any Linear layer can be expressed as Eq. (16). Note that in PyTorch, the bias is automatically expanded along rowa to  $\mathbf{b}^{p \times n}$  due to broadcasting.

$$\mathbf{Y}^{p \times n} = \mathbf{X}^{p \times m} \mathbf{W}^{m \times n} + \mathbf{b}^{1 \times n} \quad (16)$$

When solving the gradients of each node in Eq. (16), let the required back-propagation matrix be  $\mathbf{G}$ ,  $\mathbf{G} = \begin{bmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & & \vdots \\ g_{p1} & \cdots & g_{pn} \end{bmatrix} \in \mathbb{R}^{p \times n}$ , then, according to Case 1, the gradient matrices of the weight matrix  $\mathbf{W}$ , the intermediate node  $\mathbf{X}$ , and the bias  $\mathbf{b}$  can be expressed as Eq. (17).

$$\begin{aligned} \nabla_{\mathbf{W}}^{\alpha} \mathbf{Y} &= (\mathbf{X}_{ij})^T \bullet \mathbf{G} \\ \nabla_{\mathbf{X}} \mathbf{Y} &= \mathbf{G} \bullet \mathbf{W}^T \\ \nabla_{\mathbf{b}} \mathbf{Y} &= \sum_{k=1}^p \mathbf{G}_{kn} \end{aligned} \quad (17)$$

where  $\bullet$  represents matrix multiplication.  $\mathbf{X}_{ij}$  represents the fractional-order differentiation matrix for the  $(i, j)$ -th block  $\mathbf{J}^{\alpha}$ .  $\sum_{k=1}^p \mathbf{G}_{kn}$  represents summing the elements of each column for  $\mathbf{G}$  to obtain a row vector of size  $1 \times n$ , that is, the gradient of  $\mathbf{b}$ . In the Linear layer, which is also a leaf node, the weight matrix usually has a much greater impact on the output result of the ANN than the bias. Therefore, for the convenience of the qualitative discussion in the experimental part of Section 4, in Eq. (17), only the gradient matrix of  $\mathbf{W}$  is obtained through fractional-order matrix differentiation, and the gradient matrix of  $\mathbf{b}$  is obtained through integer-order matrix differentiation.

Now, if we take the first block matrix in the fractional-order Jacobian matrix as the differentiation matrix. In Eq. (17), it is  $\mathbf{X}_{11}$ . Then, the implicit calculation process of  $\nabla_{\mathbf{W}}^{\alpha} \mathbf{Y}$  can be shown as Algorithm 1.

In Algorithm 1, it mainly calculates the first and second terms of Eq. (12). According to Eq. (12), in tensor form, the first term is actually a linear mapping in integer order, and its calculation is relatively simple, corresponding to line 3 in Algorithm 1. In Case 4, many fractional-order differentiations on loss functions are also linear mappings in integer order, and they are equivalent to only calculating line 3. From line 4 to 10, it is the calculation of the second term of Eq. (12). We define the obtained

---

**Algorithm 1** Calculations of the weight matrix gradient based on  $\mathbf{X}_{11}$ 


---

**Require:**  $\mathbf{W} \in \mathbb{R}^{m \times n}$ ,  $\mathbf{X} \in \mathbb{R}^{p \times m}$ ,  $\mathbf{b} \in \mathbb{R}^{1 \times n}$ ,  $\mathbf{G} \in \mathbb{R}^{p \times n}$ ,  $\alpha$

**Ensure:**  $\mathbf{W}_{\text{grad}}$

▷  $\mathbf{W}_{\text{grad}}$  denotes  $\nabla_{\mathbf{W}}^{\alpha} \mathbf{Y}$ .

- 1:  $\mathbf{F} = \mathbf{W}[:, 0].\text{view}(1, -1) \in \mathbb{R}^{1 \times m}$     ▷ In order to obtain  $\mathbf{A}_{11}$ , the first column of  $\mathbf{W}$  is extracted and then transposed.
  - 2: Calculate the first part of  $\mathbf{W}$ , which is the first term in Eq. (12):
  - 3:  $\mathbf{W}_{\text{main}} = \mathbf{X} \odot |\mathbf{F}|^{\frac{1-\alpha}{\Gamma(2-\alpha)}} \in \mathbb{R}^{p \times m}$
  - 4: Calculate the second part of  $\mathbf{W}$ , which is the second term in Eq. (12). The calculation involves  $\mathbf{W}_{\text{partial1}}$ ,  $\mathbf{W}_{\text{partial2}}$ ,  $\mathbf{W}_{\text{partial3}}$ , and  $\mathbf{W}_{\text{partial4}}$ . Refer to Eq. (15) for details:
  - 5:  $\mathbf{W}_{\text{partial1}} = (\mathbf{X} \bullet \mathbf{F}^T).\text{view}(-1, 1) \in \mathbb{R}^{p \times 1}$
  - 6:  $\mathbf{W}_{\text{partial2}} = (\mathbf{W}_{\text{partial1}}).\text{expand}(-1, m) \in \mathbb{R}^{p \times m}$
  - 7:  $\mathbf{bias} = \text{torch.full}((p, m), \mathbf{b}[0].\text{item}()) \in \mathbb{R}^{p \times m}$     ▷ Specify  $\mathbf{b}[0]$  as a matrix of size  $p \times m$
  - 8:  $\mathbf{W}_{\text{partial3}} = (\mathbf{W}_{\text{partial2}} - \mathbf{X} \odot \mathbf{F} + \mathbf{bias}) \in \mathbb{R}^{p \times m}$
  - 9:  $\mathbf{W}_{\text{partial4}} = \text{sign}(\mathbf{F}) \odot |\mathbf{F}|^{\frac{-\alpha}{\text{gamma}(1-\alpha)}} \in \mathbb{R}^{p \times m}$
  - 10:  $\mathbf{W}_{\text{partial}} = \mathbf{W}_{\text{partial3}} \odot \mathbf{W}_{\text{partial4}} \in \mathbb{R}^{p \times m}$     ▷ Obtain the second part of  $\mathbf{W}$ . See Eqs. (12) and (15).
  - 11:  $\mathbf{W}_{\text{differentiation}} = (\mathbf{W}_{\text{main}} + \mathbf{W}_{\text{partial}})^T \in \mathbb{R}^{m \times p}$     ▷ Obtain the differentiation of  $\mathbf{W}$ .
  - 12:  $\mathbf{W}_{\text{grad}} = \mathbf{W}_{\text{differentiation}} \bullet \mathbf{G} \in \mathbb{R}^{m \times n}$     ▷ Obtain the gradient of  $\mathbf{W}$ .
- 

differentiation matrix as the fractional-order term differentiation matrix. Although the calculation of the fractional-order term differentiation matrix is relatively complex, it ensures the essential difference between the final obtained differentiation matrix and that in integer order. That is, it cannot be obtained by simply fine-tuning the learning rate or using a special learning rate scheduling strategy.

In Algorithm 1, 10 steps of operations are carried out to obtain the gradient of the weight matrix  $\mathbf{W}$ . In contrast, in integer-order differentiation,  $\nabla_{\mathbf{W}} \mathbf{Y} = \mathbf{X}^T \bullet \mathbf{G}$ . Compared with integer-order matrix differentiation, the method in this paper increases in computational complexity. However, since both are based on implicit calculation, this increase is linear and controllable. And when  $\alpha = 1.0$ , using Algorithm 1 can get

$\nabla_{\mathbf{w}}^l \mathbf{Y} = \mathbf{X}^T \bullet \mathbf{G}$ , which is the same as the result obtained by integer-order matrix differentiation.

Algorithm 1 finally obtains the gradient of the leaf node. When it replaces the gradients in the GD method and its variant methods and participates in the iterative update of the parameter matrix, it will form the corresponding fractional-order optimization method, such as the fractional-order SGD (FSGD) optimizer and the fractional-order Adam (FAdam) optimizer. Here, taking the FSGD optimizer defined in this paper as an example, we analyze its time complexity and convergence.

**Time Complexity Analysis:** Although the FSGD optimizer needs to carry out multiple steps when solving the fractional-order gradient matrix and involves a differentiation matrix with a linear mapping to integer-order and a fractional-order term differentiation matrix. However, like the SGD and Adam optimizers, their gradient matrices are based on implicit calculation, that is, the time complexity of the FSGD optimizer is the same as that of its corresponding integer-order optimization method. Here, assuming that given the network model and the *batch* of training samples, we can set the time complexity of training one *batch* as  $O(d)$ . If the total number of iterations *epoch* is  $n$ , then the total time complexity of the FSGD optimizer is  $O(n * d)$ , which is the same as that of the SGD, Adam, and FAdam optimizers.

**Convergence Analysis:** As shown in Figure 1, existing fractional-order methods improve in `Optimizer.step()`, which is an operation on the integer-order gradient matrix of the computation graph. When the operation involves a non-linear transformation of the matrix, their convergence characteristics will be different, and separate convergence analysis is required. On the contrary, the method proposed in this paper is an operation to solve the fractional-order gradient matrix of the leaf node in `Loss.backward()`. When we assume that the objective function boundary is bounded and the fractional-order gradient is bounded, their convergence characteristics are the same as those of their corresponding integer-order optimizers. The change of the fractional order has a linear impact on the convergence speed of *Loss* [37]. Therefore, there is no need to list separate formulas for proof. That is, when given the same hyperparameters, the convergence speeds of the two are the same. In many FGD methods applied to ANN training, although they have a faster convergence speed than integer order, this difference in



convergence speed is actually equivalent to the impact of different learning rates on the convergence trajectory in integer-order optimizers. In addition, in engineering applications, since training often uses an early-stopping mechanism and real-world datasets are usually biased. Therefore, the global optimum on the training set is not necessarily the optimum on the test set. In conclusion, this paper pays more attention to the performance of the network model after using FGD method training. Therefore, the relevant discussion is further carried out in the experimental part of Section 4.

## 4. Experiment

This section designs a simple ANN training task and verifies the method on two types of real-world datasets. It conducts quantitative and qualitative analyses on the performance of the method.

### 4.1. Overall Structure of the Neural Network

The neural network used in the experiment is shown in Eq. (1) and the computation graph in Figure 2. When FLinear is used to replace Linear, its overall structure during training can be summarized in Figure 3.

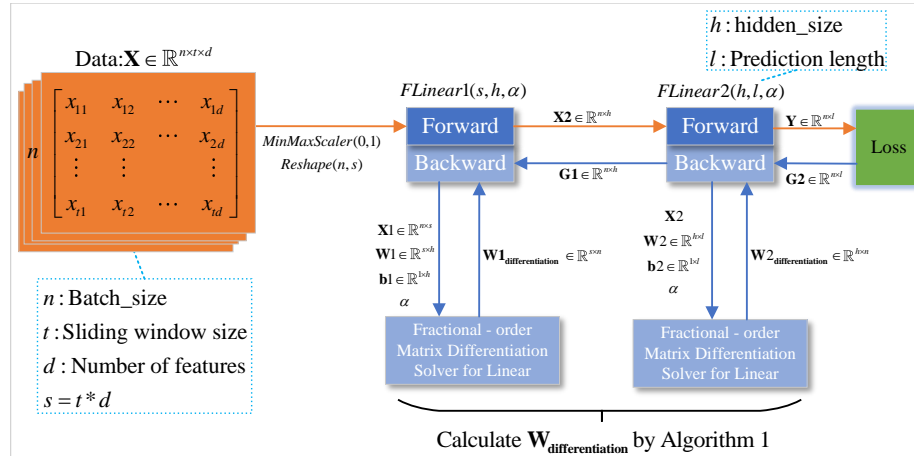


Figure 3: The overall structure of the paper.

Figure 3 is not a SOTA model. Its purpose is to facilitate the quantitative and qualitative analyses of the FGD method in this paper. It has two Linear layers and involves the problem of fractional-order matrix differentiation in the back-propagation of ANNs, meeting the requirements of the research on ANNs.

In Figure 3, FLinear only adds an incoming parameter  $\alpha$ . In Forward, FLinear is the same as Linear. Their differences lie in Backward. Compared with Linear, FLinear’s Backward has a separately designed Fractional-order Matrix differentiation Solver. The role of the solver is to calculate the fractional-order differentiation matrices of each node, and the specific calculation method corresponds to Algorithm 1. As described in the previous Section 3, the solver is the specific engineering implementation of the method in this paper.

Just like in integer-order Autograd, in Backward, the fractional-order differentiation matrix is multiplied by the back-propagated matrix to obtain the gradient matrix of the node. Then, the gradient matrices of each leaf node are saved in the computation graph for the corresponding optimizer to call and participate in the iterative update of the parameter matrix.

#### 4.2. Preparation for the Experiment

**Experimental Environment Configuration:** Since Section 4.3 involves the analysis of memory size and time consumption, the specific experimental environment configuration is introduced here. The CPU is Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz. The GPU is NVIDIA GeForce RTX 2080 Ti. The IDE is Jupyter 7.2.2. The programming language is Python 3.12.7. The deep-learning framework is PyTorch 2.6.0 + CUDA 12.6.

**Datasets and Preprocessing:** In the experiment, two real-world time-series prediction datasets are used, namely Dow Jones Industrial Average (DJI) and the public dataset ETTh1. DJI is a dataset with spikes, strong fluctuations, and strong noise, while ETTh1 is the opposite. This is to analyze the performance of the FGD method on different types of datasets. Their detailed information is shown in Table 1.

In Table 1, the five features of DJI are Open, High, Low, Volume, and Close. The Close value is predicted jointly by the five feature dimensions. The seven features

Table 1: The statistics of ETTh1 and DJI

Datasets	Features	Timesteps	Granularity	Timespan	Training:	Labels
					Validation: Test	
DJI	5	5,969	1day	Jan 3, 2000 - Sep 22, 2023	7:2:1	Close
ETTh1	7	17,420	1hour	Jul 1, 2016 - Jun 26, 2018	7:2:1	OT

of ETTh1 are HUFL, HULL, MUFL, MULL, LUFL, LULL, and OT. The OT value is predicted jointly by the seven feature dimensions. There are no outliers in either dataset, so data cleaning is not required. As shown in Figure 3, for data preprocessing, only *Min – Max* normalization and simple mini-batching are carried out.

**Hyperparameter Settings:** To fairly show the trajectories of *loss* values and quantitative metric values of different orders, some hyperparameter selections and settings are made in the experiment:

- (1) Set random seeds for batching and weight matrix initialization. When iterating at different orders, ensure that the shuffling results of batching can be reproduced and the initialized weight matrices are the same. This can eliminate the influence of randomness on the results.
- (2) During the experiment, set the learning rate uniformly as  $lr = 1e - 4$ . In many experiments on FGD method research, the influence of the learning rate on the convergence trajectory is obvious. Especially for some FGD methods, the obtained gradient matrix may actually be equivalent to providing a scalar outside the integer-order gradient matrix. This scalar changes the magnitude of the learning rate. That is, due to the change of order, the integer-order gradient matrix gets a larger or smaller learning rate for scalar matrix multiplication. Therefore, setting an appropriate learning rate can ensure that the performance display of the FGD method is not affected by the learning rate.
- (3) As shown in Figure 1, the method in this paper is to solve the fractional-order gradient matrices of each leaf node on the computation graph. During param-

After updating, the optimizer is selected according to the task requirements. In the whole experiment process of this paper, the SGD optimizer is selected, that is, the FSGD optimizer. Compared with other optimizers (such as the popular Adam), the SGD optimizer will not change the characteristics of the parameter matrix. Other optimizers will perform various non-linear operations on the gradient matrix after obtaining the leaf node gradient matrix, destroying the additivity and multiplicativity of the gradient matrix, thus changing the characteristics of the fractional-order gradient matrix. Finally, it will affect our qualitative and quantitative analysis of the FGD method. Similarly, for the parameter settings of the SGD optimizer, we set  $weight\_decay = 0$  and  $momentum = 0$ . Because these two hyperparameters will also destroy the additivity of the fractional-order gradient matrix.

- (4) Similarly, the ANN used in the experiment does not set an activation function. Because the activation function will also destroy the additivity and multiplicativity of the fractional-order gradient matrix, affecting our qualitative and quantitative analysis of the results.
- (5) According to Figure 3, set these hyperparameters uniformly: the sliding window size is 36,  $Batch\_size = 256$ , the prediction length is 48, and the loss function uses MSELoss. According to the characteristics of DJI and ETTh1, to better show the convergence trajectories of different orders, when training DJI, set  $hidden\_size = 256$ . When training ETTh1, set  $hidden\_size = 128$ .

#### 4.3. Experimental Results Performance Analysis

We respectively train the training sets of two datasets and display their convergence trajectories at 1500 iterations. Meanwhile, after each iteration, the model is used for quantitative evaluation on the validation set, with the evaluation metric being MSE, i.e., MSELoss. Experiments with different orders are carried out, and the results of orders  $\alpha = \{0.7, 0.8, 0.9, 1.0\}$  are collected and shown in Figure 4.

Overall, Figure 4 shows the characteristics of the FSGD optimizer that are different from those of the SGD optimizer, including the self-contained learning rate scheduling

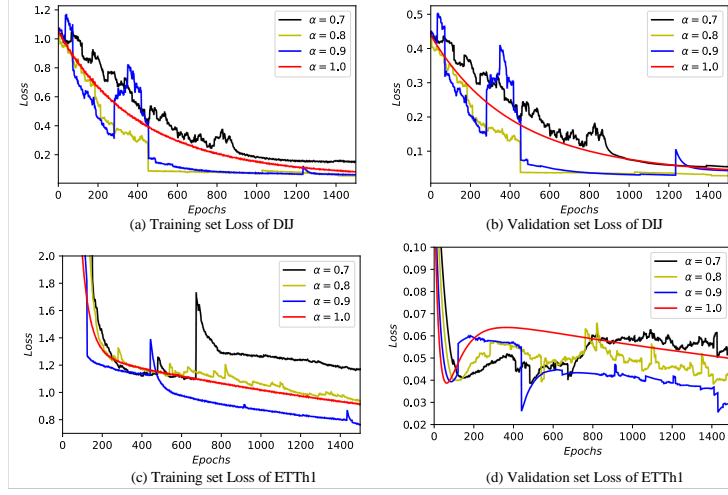


Figure 4: The convergence curves of DJI and ETTh1 on the training and validation sets.

strategy and regularization. The details are as follows:

- (1) Although there are various interfering factors, it can still be seen that, at most fractional orders, due to the influence of the learning rate scheduling strategy of the FSGD optimizer, the *Loss* of the two datasets often decreases faster than that of the integer order in the early stage of iterations. Of course, this difference in convergence speed is at a linear level and has not reached the change level from the SGD optimizer to the Adam optimizer.
- (2) It can also be seen in Figure 4(d) that, during the fractional-order optimization process, due to the special regularization characteristic of the FSGD optimizer, even in the case of overfitting, they can jump out of the local optimal solution and search for the global optimal solution. In contrast, once the integer order overfits and falls into a local optimal solution, it cannot jump out.
- (3) Overall, compared with the SGD optimizer, the convergence trajectory of the FSGD optimizer is not smooth. This is caused by the fractional-order term in Eq. (12). When performing fractional-order matrix differentiation, even though this method only extracts a block matrix from the Jacobian matrix differentiation, the fractional-order differentiation matrix of the weight matrix is affected by the

input matrix and the bias vector. In ANNs, because of the early-stopping mechanism, we usually only save the model with the smallest *Loss* on the validation set. This non-smooth characteristic on the convergence trajectory will not have a great impact on performance.

- (4) Since we finally obtain the model at the moment when the *Loss* on the validation set is the smallest, by comparing the convergence trajectories of different orders in Figure 4(b) and (d), it can be found that the performance of  $\alpha = 0.7$  is generally worse than that of the integer order, and  $\alpha = 0.8$  is the second. By analyzing the fractional-order differentiation term in Eq. (12), it can be seen that when the value of  $\alpha$  approaches 0 more and more, the value of the differentiation will become more and more extreme. From the perspective of the FSGD optimizer, it means gradient explosion. The training of the ANN model has almost no opportunity to give full play to the unique advantages of the FSGD optimizer and will become worse and worse due to gradient explosion. This also verifies the results of some previous studies [4, 7, 14], that is, the optimal fractional-order selection of the FGD method is usually  $0.9 < \alpha < 1.0$ .

In Figure 4(b) and (d), the convergence trajectory of *Loss* can already well show the performance of each order. However, for quantitative analysis, we use the optimal models to make quantitative evaluations on the test set. In addition to MSE, the mean absolute error (MAE) is also used as the evaluation metric. They are commonly used evaluation metrics in regression tasks, and a smaller value indicates better performance. The detailed quantitative results are shown in Figure 5.

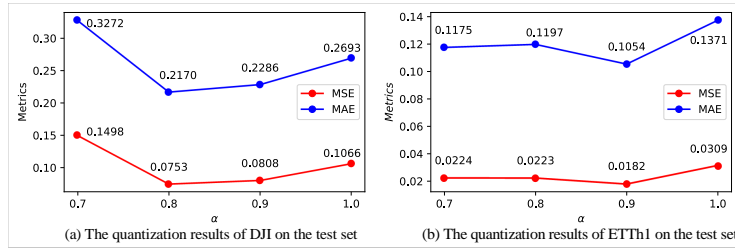


Figure 5: The test set metrics on different  $\alpha$ .

Overall in Figure 5, the performance of the fractional-order orders  $\alpha = \{0.8, 0.9\}$  is better. This verifies the results and speculations of the validation set. In addition, in Figure 5(a), the order change is more sensitive to the results. As mentioned before, DJI has strong noise. This means that it needs more regularization to prevent overfitting. The FSGD optimizer has self-contained regularization and will work better for DJI-type datasets. Therefore, after selecting an appropriate order, the performance improvement will be obvious. Similarly, the results in Figure 5(b) are reverse verification. Since ETTh1 is a stationary dataset with little noise, the change in the quantized results is not as large as that of DJI when the order changes. Finally, we can see that in DJI, the results of  $\alpha = 0.7$  deteriorate rapidly. It is the only fractional order with worse performance than the integer order. This verifies the analysis in the fourth point of Figure 4.

**Analysis of Time Consumption and GPU Memory Usage:** To analyze the time consumption and GPU memory usage of the FSGD optimizer (in this paper, it refers to SGD + FLinear) and the integer-order optimizer, we record the running time and GPU memory at different orders for the first epoch of ETTh1’s training set. Their detailed information is shown in Table 2.

Table 2: The comparison of time consumption and GPU Memory usage at different orders.

Method	$\alpha$	Time (s)	Memory (MB)
FSGD	0.9	0.7821	38.4521
FSGD	1.0	0.5873	38.4521
SGD	#	0.3041	38.4521

In Table 2, it can be found that whether directly using the SGD optimizer, and using integer orders or fractional orders in the FSGD optimizer, their GPU memory usage is the same. As known from Figure 1 and 3, the FSGD optimizer (including the FAdam optimizer) actually changes the calculation method of the gradient matrix of each node on the computation graph. Given the model and data, the size of the computation graph is fixed and will not increase the GPU memory due to the addition of a fractional-order matrix differentiation solver in *Loss.Backward(retain\_graph = True)*.

In Table 2, we focus on analyzing the time consumption under three conditions.

First, although the time consumptions of the three are different, they still change at a linear level. This verifies the time-complexity analysis of Algorithm 1. Second, the FSGD optimizer takes more time than the SGD optimizer. This is because calculating the fractional-order matrix differentiation requires more intermediate processes, and a certain amount of time needs to be consumed in these processes. Third, in the FSGD optimizer, the time consumption of  $\alpha = 0.9$  is more than that of  $\alpha = 1.0$ . Through the discussion of the previous formula 12 and Algorithm 1, it can be known that the training result of the FSGD optimizer with  $\alpha = 1.0$  is equal to that of the SGD optimizer, but compared with  $\alpha = 0.9$ ,  $\alpha = 1.0$  can eliminate the calculation of the fractional-order differentiation term, thus saving most of the calculations in the fractional-order matrix differentiation solver and reducing time consumption.

Through the display and analysis of Figures 4 and 5 and Table 2, we can verify the performance of the FSGD optimizer, and then verify the superiority of the fractional-order matrix differentiation theory, as well as the feasibility of  $\mathbf{J}^\alpha$  in engineering applications.

## 5. Summary

This paper studies the fractional-order matrix differentiation method in ANNs. By calculating the fractional-order differentiation matrix using the  $\mathbf{J}^\alpha$  method, it proposes a novel solution for the research of the FGD method, solving the problem that the FGD method cannot be applied inside ANNs. The feasibility of the method is verified from both theoretical and experimental perspectives, laying a foundation for further research and improvement of the FAutograd technology. In practical engineering applications, by replacing Linear with FLinear, the optimizer becomes a fractional-order optimizer. For example, in the experiments, when the optimizer during training is changed from SGD to Adam, the FSGD optimizer becomes the FAdam optimizer. However, as described in the hyperparameter setting (3), adaptive learning rate optimizers will change the original characteristics of the fractional-order gradient matrix, affecting our qualitative and quantitative research on  $\mathbf{J}^\alpha$ . Therefore, the FAdam optimizer is not selected in the experimental part.



In future work, we will further study and improve FLinear. Moreover, we will reconstruct modules such as CNN, RNN, and Transformer into their corresponding fractional-order modules, namely FCNN, FRNN, and FTransformer. Finally, we will complete the construction of the FAutograd framework to enable the application of fractional-order matrix differentiation in more types of ANNs.

From the discussion in the experimental part, we can find that in practical engineering applications, the FGD method is more prone to gradient explosion than the GD method. This situation is more obvious when the fractional order is extreme settings. In some cases, the FGD method will experience performance degradation due to the gradient explosion problem. We know that this is caused by the fractional-order differentiation term. Therefore, in the practical engineering applications of ANNs, we will start from the fractional-order symbolic differentiation formula and further optimize it. This is our future research direction.

## Appendix A Convergence Trajectories and Gradient Directions

Let the scalar function  $y = (x_1 + 2)^2 + (x_2 + 3)^2$ . It can be known that the extreme point of the function is  $(x_1, x_2) = (-2, -3)$ . According to Eq. (12), the update formulas of the two parameters are as follows:

$$x_1^{(k+1)} = x_1^{(k)} - \eta \bullet \text{sign}(x_1^{(k)} + 2) \left( \frac{2}{\Gamma(3-\alpha)} |x_1^{(k)} + 2|^{2-\alpha} + \frac{(x_2^{(k)} + 3)^2}{\Gamma(1-\alpha)} |x_1^{(k)} + 2|^{-\alpha} \right) \quad (\text{A.1})$$

$$x_2^{(k+1)} = x_2^{(k)} - \eta \bullet \text{sign}(x_2^{(k)} + 3) \left( \frac{(x_1^{(k)} + 2)^2}{\Gamma(1-\alpha)} |x_2^{(k)} + 3|^{-\alpha} + \frac{2}{\Gamma(3-\alpha)} |x_2^{(k)} + 3|^{2-\alpha} \right) \quad (\text{A.2})$$

where  $\eta$  is the learning rate and  $k = 1, 2, \dots, N$ .

From Formulas (A.1) and (A.2), the convergence trajectories of the parameters during iterations can be obtained. When  $\alpha = \{0.5, 1.0\}$ , we randomly set  $(x_1^0, x_2^0) = (-3.5, -4.7)$  (which are also the lower bounds of their respective fractional-order differentiations) and record the results of the first 20 iterations. See Figure A.1.

In Figure A.1, it can be seen that compared with integer-order differentiation, the convergence trajectory of fractional-order differentiation shows the characteristic of being fast at first and then slow. Usually, in deep-learning optimization methods, such an effect can only be achieved by calling a learning rate scheduler. Therefore, fractional-

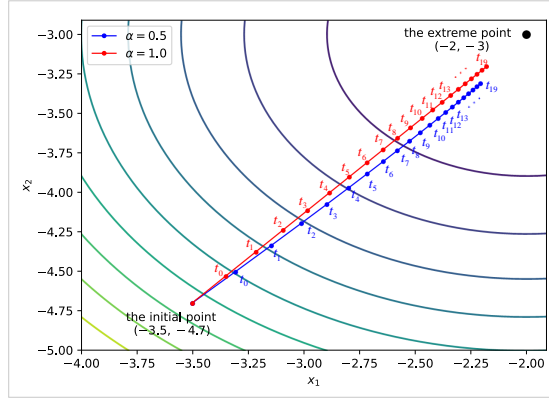


Figure A.1: The convergence trajectories of the parameters in the fractional order and the integer order.

order differentiation is actually equivalent to having a special learning rate scheduling strategy.

Regarding the characteristics of fractional-order at saddle points, given a ternary function  $z = x^2 - y^2$ , draw its three-dimensional graph. When the parameters are at the saddle point, we solve the gradients respectively using integer-order differentiation and fractional-order differentiation, that is, using the GD method and the FGD method respectively. See Figure A.2.

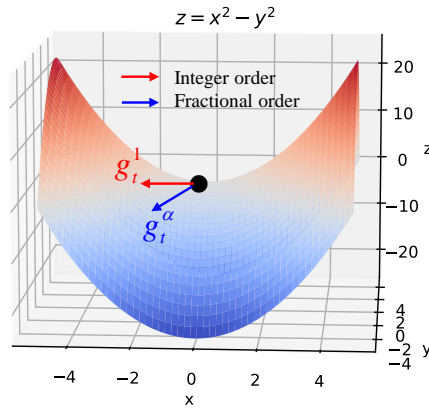


Figure A.2: The gradient directions at the saddle points for the fractional order and the integer order.

As is well known, integer-order differentiation will cause the parameters to oscillate

left and right at the saddle point. On the contrary, according to Formula 12 and Figure A.2, due to the influence of the fractional-order differentiation term, when  $\alpha \neq 1.0$ , the gradient vector must not be parallel to the  $xz$  plane. Therefore, the FGD method will not get stuck at the saddle point.

## Appendix B Proof of Fractional-order Differentiation Regularization in Artificial Neural Networks

*Proof.* Let  $J(w)$  be the objective function. For the GD method, we have a regularization method with a penalty function. That is,

$$\tilde{J}(w) = J(w) + p(w) \quad (\text{B.1})$$

where  $\tilde{J}(w)$  is the total objective function and  $p(w)$  is the penalty function. Specifically,  $L^1$  and  $L^2$  are penalty functions when  $p(w)$  takes specific forms.

When using integer order to calculate the partial derivative of  $\tilde{J}(w)$ , we have

$$\tilde{J}'(w) = J'(w) + p'(w) = J'(y)y'(x) + p'(w) \quad (\text{B.2})$$

In contrast, for an objective function without a penalty function, using fractional-order differentiation and considering  $y = wx + b$ , we have

$$\begin{aligned} J'(w) &= J'(y) \frac{\partial^\alpha y}{\partial w^\alpha} = J'(y) \left( \frac{x}{\Gamma(2-\alpha)} |w|^{1-\alpha} + \text{sign}(w) \frac{b}{\Gamma(1-\alpha)} |w|^{-\alpha} \right) \\ &= J'(y) \frac{x}{\Gamma(2-\alpha)} |w|^{1-\alpha} + J'(y) \text{sign}(w) \frac{b}{\Gamma(1-\alpha)} |w|^{-\alpha} \end{aligned} \quad (\text{B.3})$$

In Eq. (B.3), let  $J'_1(w) = J'(y) \frac{x}{\Gamma(2-\alpha)} |w|^{1-\alpha}$  and  $p'_1(w) = J'(y) \text{sign}(w) \frac{b}{\Gamma(1-\alpha)} |w|^{-\alpha}$ . Then, according to Eq. (B.2), we have

$$J'(w) = J'_1(w) + p'_1(w) \quad (\text{B.4})$$

From Eq. (B.4), it can be seen that when using fractional-order differentiation, the objective function is equivalent to having a built-in penalty function. Therefore, fractional-order differentiation has the characteristic of being equivalent to having built-in regularization. Moreover, this kind of regularization is not a linear mapping with  $L^1$  and  $L^2$ .  $\square$

## References

- [1] Hongying Yang, Shuren Qi, Jialin Tian, Panpan Niu, and Xiangyang Wang. Robust and discriminative image representation: Fractional-order jacobi-fourier moments. *Pattern Recognition*, 115:107898, 2021.
- [2] Sangheeta Roy, Palaiahnakote Shivakumara, Hamid A Jalab, Rabha W Ibrahim, Umapada Pal, and Tong Lu. Fractional poisson enhancement model for text detection and recognition in video frames. *Pattern Recognition*, 52:433–447, 2016.
- [3] Tengfei Yang, Zhiquan Liu, Jingjing Guo, Yong Yu, Fang Ren, and Teng Wang. Image analysis by fractional-order weighted spherical bessel-fourier moments. *Pattern Recognition*, 157:110872, 2025.
- [4] Yong Wang, Yuli He, and Zhiguang Zhu. Study on fast speed fractional order gradient descent method and its application in neural networks. *Neurocomputing*, 489:366–376, 2022.
- [5] Zeshan Aslam Khan, Naveed Ishtiaq Chaudhary, and Syed Zubair. Fractional stochastic gradient descent for recommender systems. *Electronic Markets*, 29: 275–285, 2019.
- [6] Yuquan Chen, Qing Gao, Yiheng Wei, and Yong Wang. Study on fractional order gradient methods. *Applied Mathematics and Computation*, 314:310–321, 2017.
- [7] Xingwen Zhou, Zhenghao You, Weiguo Sun, Dongdong Zhao, and Shi Yan. Fractional-order stochastic gradient descent method with momentum and energy for deep neural networks. *Neural Networks*, 181:106810, 2025.
- [8] Weipu Lou, Wei Gao, Xianwei Han, and Yimin Zhang. Variable order fractional gradient descent method and its application in neural networks optimization. In *2022 34th Chinese Control and Decision Conference (CCDC)*, pages 109–114. IEEE, 2022.
- [9] Xiaojun Zhou, Chunna Zhao, and Yaqun Huang. A deep learning optimizer based on grünwald–letnikov fractional order definition. *Mathematics*, 11(2):316, 2023.

- [10] ZhongLiang Yu, Guanghui Sun, and Jianfeng Lv. A fractional-order momentum optimization approach of deep neural networks. *Neural Computing and Applications*, 34(9):7091–7111, 2022.
- [11] Tao Kan, Zhe Gao, Chuang Yang, and Jing Jian. Convolutional neural networks based on fractional-order momentum for parameter training. *Neurocomputing*, 449:85–99, 2021.
- [12] Xuetao Xie, Yi-Fei Pu, and Jian Wang. A fractional gradient descent algorithm robust to the initial weights of multilayer perceptron. *Neural Networks*, 158:154–170, 2023.
- [13] Sroor M Elnady, Mohamed El-Beltagy, Ahmed G Radwan, and Mohammed E Fouda. A comprehensive survey of fractional gradient descent methods and their convergence analysis. *Chaos, Solitons & Fractals*, 194:116154, 2025.
- [14] Xiaojun Zhou, Chunna Zhao, Yaqun Huang, Chengli Zhou, and Junjie Ye. Improved fractional-order gradient descent method based on multilayer perceptron. *Neural Networks*, 183:106970, 2025.
- [15] Min-Rong Chen, Bi-Peng Chen, Guo-Qiang Zeng, Kang-Di Lu, and Ping Chu. An adaptive fractional-order bp neural network based on extremal optimization for handwritten digits recognition. *Neurocomputing*, 391:260–272, 2020.
- [16] Hongqiu Zhu, Zhiliang Wu, Chunhua Yang, Tao Peng, Zhiwen Chen, and Xiaoyue Yang. Fractional steepest ascent method for tcu fault detection. *IFAC-PapersOnLine*, 51(24):1336–1342, 2018.
- [17] Bi-Peng Chen, Yun Chen, Guo-Qiang Zeng, and Qingshan She. Fractional-order convolutional neural networks with population extremal optimization. *Neurocomputing*, 477:36–45, 2022.
- [18] Abdul Wahab, Shujaat Khan, Imran Naseem, and Jong Chul Ye. Performance analysis of fractional learning algorithms. *IEEE Transactions on Signal Processing*, 70:5164–5177, 2022.

- [19] E Viera-Martin, JF Gómez-Aguilar, JE Solís-Pérez, JA Hernández-Pérez, and RF Escobar-Jiménez. Artificial neural networks: a practical review of applications involving fractional calculus. *The European Physical Journal Special Topics*, 231(10):2059–2095, 2022.
- [20] Manisha Joshi, Savita Bhosale, and Vishwesh A Vyawahare. A survey of fractional calculus applications in artificial neural networks. *Artificial Intelligence Review*, pages 1–54, 2023.
- [21] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [22] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.
- [23] Yu. Nesterov. A method of solving a convex programming problem with convergence rate  $O(1/k^2)$ . In *Soviet Mathematics Doklady*, 1983.
- [24] Richard S Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 8, 1986.
- [25] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.
- [26] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [27] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- [28] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

- [29] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.
- [30] Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*, 2018.
- [31] Yeonjong Shin, Jérôme Darbon, and George Em Karniadakis. Accelerating gradient descent and adam via fractional gradients. *Neural Networks*, 161:185–201, 2023.
- [32] Jian Wang, Yanqing Wen, Yida Gou, Zhenyun Ye, and Hua Chen. Fractional-order gradient descent learning of bp neural networks with caputo derivative. *Neural networks*, 89:19–30, 2017.
- [33] Yuan Cao and Shuai Su. Fractional gradient descent algorithms for systems with outliers: A matrix fractional derivative or a scalar fractional derivative. *Chaos, Solitons & Fractals*, 174:113881, 2023.
- [34] Haixin Wu, Yaqian Mao, Jiacheng Weng, Yue Yu, and Jianhong Wang. Fractional light gradient boosting machine ensemble learning model: A non-causal fractional difference descent approach. *Information Fusion*, 118:102947, 2025.
- [35] Shuli Yan, Qi Su, Zaiwu Gong, Xiangyan Zeng, and Enrique Herrera-Viedma. Online public opinion prediction based on rolling fractional grey model with new information priority. *Information Fusion*, 91:277–298, 2023.
- [36] Igor Podlubny. *Fractional differential equations: an introduction to fractional derivatives, fractional differential equations, to methods of their solution and some of their applications*. Elsevier, 1998.
- [37] Priyanka Harjule, Rinki Sharma, and Rajesh Kumar. Fractional-order gradient approach for optimizing neural networks: A theoretical and empirical analysis. *Chaos, Solitons & Fractals*, 192:116009, 2025.

### **Declaration of interests**

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: