RELU-KAN: NEW KOLMOGOROV-ARNOLD NETWORKS THAT ONLY NEED MATRIX ADDITION, DOT MULTIPLICATION, AND RELU *

Qi Qiu
School of Computer Science
University of South China
Hengyang, Hunan, China
qiuqi@stu.usc.edu.cn

Tao Zhu
School of Computer Science
University of South China
Hengyang, HuNan, China
tzhu@usc.edu.cn

ABSTRACT

Limited by the complexity of basis function (B-spline) calculations, Kolmogorov-Arnold Networks (KAN) suffer from restricted parallel computing capability on GPUs. This paper proposes a novel ReLU-KAN implementation that inherits the core idea of KAN. By adopting ReLU (Rectified Linear Unit) and point-wise multiplication, we simplify the design of KAN's basis function and optimize the computation process for efficient CUDA computing. The proposed ReLU-KAN architecture can be readily implemented on existing deep learning frameworks (e.g., PyTorch) for both inference and training. Experimental results demonstrate that ReLU-KAN achieves a 20x speedup compared to traditional KAN with 4-layer networks. Furthermore, ReLU-KAN exhibits a more stable training process with superior fitting ability while preserving the "catastrophic forgetting avoidance" property of KAN. You can get the code in https://github.com/quiqi/relu_kan

***K*eywords** Kolmogorov-Arnold Networks · Parallel Computing · Rectified Linear Unit

1 Introduction

Kolmogorov-Arnold Networks (KANs) [1] have recently garnered significant interest due to their exceptional performance and novel architecture [2, 3]. Researchers have rapidly adopted KANs for tackling diverse problems [4, 5]. However, a key challenge hindering their wider adoption is the limitation in leveraging GPUs' parallel processing capabilities. This bottleneck stems from the inherent complexity of KANs' spline function design, ultimately impacting processing speed and scalability.

This paper introduces a simplified basis function:

$$R_i(x) = [\mathrm{ReLU}(e_i - x) \times \mathrm{ReLU}(x - s_i)]^2 \times 16/(b_i - a_i)^4 \tag{1}$$

where ReLU(x)=max(0, x) [6] and optimizes KAN operations for efficient GPU parallel computing based on this simplified basis function. Firstly, we express the entire basis function computation as matrix operations to fully utilize GPU's parallel processing capabilities. Secondly, similar to positional encoding in Transformer[cite], we pre-generate non-trainable parameters to accelerate computation. Finally, we represent the weighted sum of basis functions as a convolution operation, enabling the new KAN architecture to be easily implemented on existing deep learning frameworks. We implemented the core code for the KAN architecture using PyTorch with less than 30 lines of code. The new KAN architecture is called ReLU-KAN in this paper.

We evaluated ReLU-KAN's performance on a set of functions used in the original KAN paper. Compared to KAN, ReLU-KAN demonstrates significant improvements in training speed, convergence stability, and fitting accuracy,

* *Citation*:

particularly for larger network architectures. Notably, ReLU-KAN inherits most of KAN's key properties, including hyperparameters like the number of grids and its ability to prevent catastrophic forgetting.

Specifically, in the existing experiments, ReLU-KAN is 5 to 20 times faster than KAN in training, and the accuracy of ReLU-KAN is 2 orders of magnitude higher than KAN.

This paper presents the following key contributions:

- **A Simplified Basis Function:** We introduce a simplified basis function, R(x), which retains the fitting capability of the original KAN basis function while offering enhanced efficiency.
- **Matrix-Based KAN Operations:** Building upon the simplified basis function, we optimize KAN operations for efficient matrix computations. This optimization enables better compatibility with GPU processing and facilitates implementation within existing deep learning frameworks.

In the following sections, we delve into the details of our contributions: In section 2, We introduce KAN, conceptualizing it as an extension of Multilayer Perceptrons (MLPs). We provide a high-level overview of KAN and explore potential approaches for constructing similar network architectures; in section 3, we present the ReLU-KAN architecture, highlighting its core components and efficient PyTorch implementation; in section 4, we conduct comprehensive experiments to evaluate ReLU-KAN's performance against KAN. We will explore ReLU-KAN's advantages in training speed, convergence stability, and fitting accuracy, particularly for larger networks. Furthermore, we will confirm ReLU-KAN's ability to prevent catastrophic forgetting.

2 Related Work

This section provides an overview of Kolmogorov-Arnold Networks (KANs). As our work primarily focuses on improving KAN's basis function, we will delve deeper into the role of the B-spline function within the KAN architecture.

2.1 Kolmogorov-Arnold Networks

The Kolmogorov-Arnold representation theorem confirms that a high-dimensional function can be represented as a composition of a finite number of one-dimensional functions as Eq 2.

$$f(\boldsymbol{x}) = \sum_{i=1}^{2n+1} \Phi_i(\sum_{j=1}^{n} \phi_{i,j}(x_j)) \tag{2}$$

where $\phi_{i,j}$ is called the inner function and Φ_i is called the outer function. Building upon the theorem's mathematical framework, the original KAN paper presented a two-layer structure for fitting high-dimensional functions and relaxes the restriction on the number of functions in multi-layer KAN networks.

Let's consider a KAN network where the input vector, denoted by $\boldsymbol{x}$, has a length of m. The output vector $\boldsymbol{y}$ has a length of n, The following equation describes the corresponding KAN layer as Eq 3

$$\boldsymbol{y} = \begin{pmatrix} \Phi(\cdot)_1 \\ \Phi(\cdot)_2 \\ \vdots \\ \Phi(\cdot)_n \end{pmatrix} \cdot \begin{pmatrix} \phi(\cdot)_{11} & \phi(\cdot)_{12} & \cdots & \phi(\cdot)_{1m} \\ \phi(\cdot)_{21} & \phi(\cdot)_{22} & \cdots & \phi(\cdot)_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\cdot)_{n1} & \phi(\cdot)_{n2} & \cdots & \phi(\cdot)_{nm} \end{pmatrix} \boldsymbol{x} \tag{3}$$

In order to ensure the representation power of $\phi_{ij}$and Φ_i, they are represented as linear combinations of multiple B-spline functions and a bias function as Eq 4:

$$\phi(x) = w_b x/(1 + e^{-x}) + w_s \sum c_i B_i(x) \tag{4}$$

where $B_i(x)$ is a B-spline function.

If we set $\phi_{ij}(x_j) = w_{ij}x_j, \Phi_i(x) = \text{ReLU}(x)$, KAN is equivalent to a single-layer MLP. In this sense, KAN can be thought of as an extension of MLP.

The activation function plays a crucial role in MLP because the $\phi_{ij}(x_j) = w_{ij}x_j$ lacks nonlinear fitting ability. But if $\phi_{ij}(x)$ is a nonlinear function, the activation function can be omitted and the KAN network can be represented as Eq 5:

$$\boldsymbol{y} = \begin{pmatrix} \phi(\cdot)_{11} & \phi(\cdot)_{12} & \cdots & \phi(\cdot)_{1m} \\ \phi(\cdot)_{21} & \phi(\cdot)_{22} & \cdots & \phi(\cdot)_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\cdot)_{n1} & \phi(\cdot)_{n2} & \cdots & \phi(\cdot)_{nm} \end{pmatrix} \boldsymbol{x} \tag{5}$$

we can construct more KAN-like structures based on Eq 5, just by finding suitable nonlinear $\phi(x)$.

2.2 B-splines

In KAN network, a set of B-spline functions denoted as $\boldsymbol{B} = \{B_1(a_1, k, s, x), B_2(a_2, k, s, x), \ldots, B_n(a_n, k, s, x)\}$ are used as basis functions to represent any unary function on a finite domain. These B-spline functions share the same shape but have different positions. Each term $B_i(a_i, k, x)$ is a bell-shaped function, and a_i, k, and s are the hyper-parameters of B_i. The a_i is used to control the position of the symmetry axis, k determines the range of the non-zero region, and s is the unit interval. The i spline B_i is shown as Fig 1(Let's say $k = 3$).

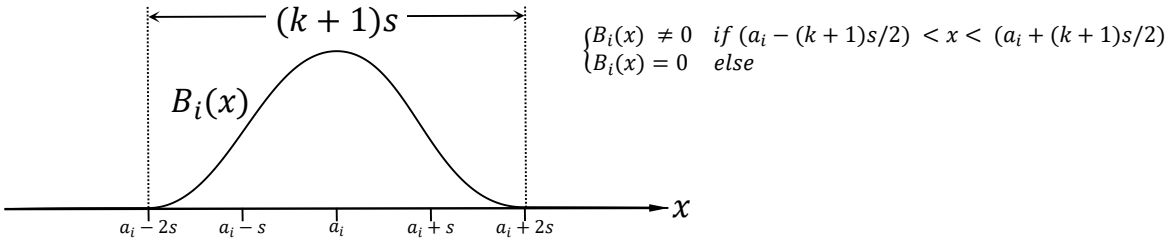


Figure 1: The i^{th} B-spline.

The hyperparameters of basis function set $\boldsymbol{B}$ depend on the number of grids, denoted by G. Specifically, when the domain of the function to be approximated is $x \in [0, 1]$, we have $n = G + k$ basis functions, step size: $s = 1/G$ and $a_i = \frac{2i+1-k}{2G}$. Figure 2 illustrates the appearance of $\boldsymbol{B}$ for the case of $G = 5$ and $k = 3$.

The function to be fitted $f(x)$ is expressed as Eq 4 in KAN. Using an optimization algorithm such as gradient descent to determine the values of w_b, w_s, and $\boldsymbol{c} = [c_1, c_2, \ldots, c_n]$, we obtain $\phi(x)$ fitted using B-splines function.

Increasing the number of grids, G, leads to a greater number of trainable parameters, consequently enhancing the model's fitting ability. However, a larger k value strengthens the coupling between the B-spline functions, which can also improve fitting ability. As both G and k are effective hyper parameters for controlling the model's fitting capability, we and retain them within the ReLU-KAN architecture.

The spline function $B_i(x)$is a very complex piecewise function, so the solving process of the spline function cannot be characterized as a matrix operation, so it cannot make full use of the parallel ability of GPU.

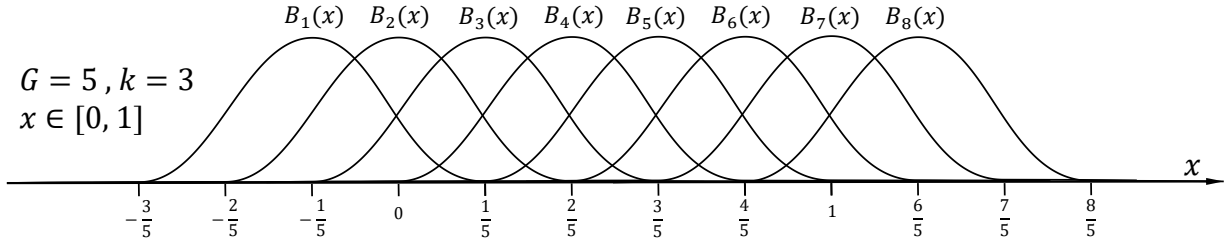


Figure 2: Appearance of $\boldsymbol{B}$ for the case of $G = 5$ and $k = 3$.

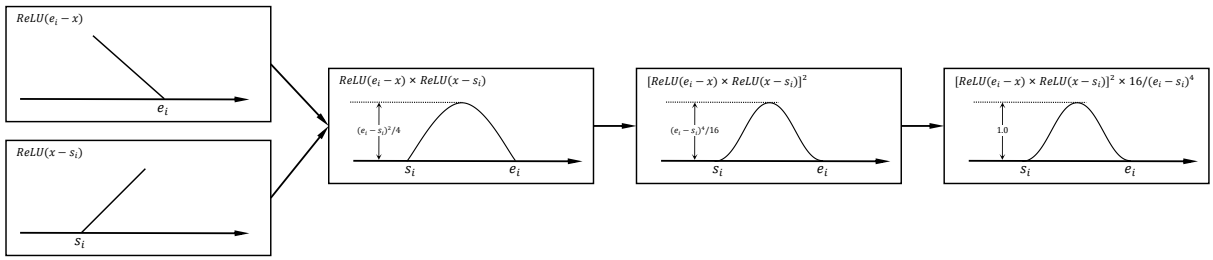


Figure 3: The construction of R_i

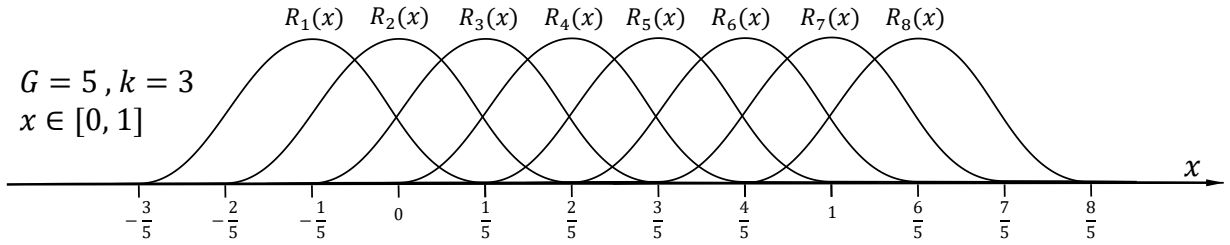


Figure 4: Appearance of $\boldsymbol{R}$ for the case of $G = 5$ and $k = 3$.

3 Methods

3.1 ReLU-KAN

We use the simpler function $R_i(x)$ to replace the B-spline function in KAN as the new basis function:

$$R_i(x) = [\text{ReLU}(e_i - x) \times \text{ReLU}(x - s_i)]^2 \times 16/(e_i - s_i)^4 \tag{6}$$

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

Like $B_i(x)$, $R_i(x)$ is also a unary bell-shaped function, which is nonzero at $x \in [s_i, e_i]$and zero at other intervals. The $\text{ReLU}(x)$ function is used to limit the range of nonzero values, and the squaring operation is used to increase the smoothness of the function. $16/(e_i - s_i)^4$ for normalization. As Fig 3

Multiple basis function R_i can form the basis function set $\boldsymbol{R} = \{R_1(x), R_2(x), \ldots, R_n(x)\}$, $\boldsymbol{R}$ inherited many properties of $\boldsymbol{B}$, It is again composed of $n$basis functions with the same shape but different positions, and the number of basis functions n and a_i, b_i are also determined by the number of grids $G$and the span parameter k.

A set of basis functions, denoted by $\boldsymbol{R} = \{R_1(x), R_2(x), \ldots, R_n(x)\}$, can be constructed from multiple basis functions, R_i. And $\boldsymbol{R}$ inherits many properties from the $\boldsymbol{B}$, $\boldsymbol{R}$ consists of n basis functions with identical shapes but varying positions. The number of basis functions, n, along with the position parameters a_i and b_i are still determined by the number of grids, G, and the span parameter, k.

If we assume that the domain of the function to be fitted is $x \in [0, 1]$, the number of grids is G, and the span parameter is k, the number of spline functions is $n = G + k$. The parameter of $R_i(x)$, $s_i = \frac{i-k-1}{G}, e_i = \frac{i}{G}$.

For example, the Fig 4 shows a schematic representation of $\boldsymbol{R}$ for $G = 5$ and $k = 3$.

The ReLU-KAN layer can also be expressed by Eq(5), and the corresponding $\phi(x)$ of ReLU-KAN removes the bias function and is further simplified to Eq 7.

$$\phi(x) = \sum_{i=1}^{G+k} w_i R_i(x) \tag{7}$$

The multi-layer ReLU-KAN can be represented as the Fig 5 In the following expression, we use $[n_1, n_2, \ldots, n_k]$to represent a ReLU-KAN of $k - 1$ layers, and the i layer takes as input the output of $i - 1$ layers. Its input vector is of length n_i and its output vector is of length n_{i+1}.

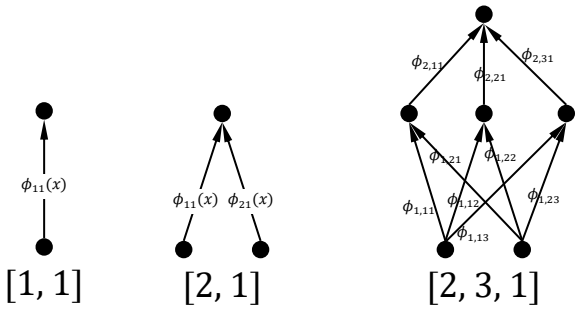


Figure 5: The multi-layer ReLU-KAN

3.2 Operational Optimization

Given the hyper parameters G and k, the number of inputs m, and the number of outputs n, we pre-compute the vectors $\boldsymbol{s}$ and $\boldsymbol{e}$ as Eq 8

$$\boldsymbol{s} = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,G+k} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,G+k} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m,1} & s_{m,2} & \cdots & s_{m,G+k} \end{pmatrix} \quad \boldsymbol{e} = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,G+k} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,G+k} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m,1} & e_{m,2} & \cdots & e_{m,G+k} \end{pmatrix} \tag{8}$$

where, $s_{i,j} = \frac{j-k-1}{G}, e_{i,j} = \frac{j}{G}$.

When using Eq 6 as the basis function, we define a normalization constant $r = \frac{16G^4}{(k+1)^4}$, We also define a convolutional operation C with one input channel, n output channels, and a kernel size of $m \times (G+k)$. With these definitions, Eq 5 can be decomposed into the following matrix operation:

$$\begin{aligned} \boldsymbol{x}_1 &= \text{ReLU}(\boldsymbol{e} - \boldsymbol{x}) \\ \boldsymbol{x}_2 &= \text{ReLU}(\boldsymbol{x} - \boldsymbol{s}) \\ \boldsymbol{x}_3 &= r \times \boldsymbol{x}_1 \cdot \boldsymbol{x}_2 \\ \boldsymbol{x}_4 &= \boldsymbol{x}_3 \cdot \boldsymbol{x}_3 \\ \boldsymbol{y} &= C(\boldsymbol{x}_4) \end{aligned}$$

In order to describe the above process more clearly, we give the Python code of the ReLU-KAN layer based on PyTorch Code 1. It is very simple code and does not need to take up too much space.

Listing 1: Example Python Code

```
import numpy as np
import torch
import torch.nn as nn

class ReLUKANLayer(nn.Module):
    def __init__(self, input_size: int, g: int, k: int, output_size: int):
        super().__init__()
        self.g, self.k, self.r = g, k, 16*g*g*g*g / ((k+1)*(k+1)*(k+1)*(k+1))
        self.input_size, self.output_size = input_size, output_size
        phase_low = np.arange(-k, g) / g
        phase_height = phase_low + (k+1) / g
        phase_low = np.array([phase_low for i in range(input_size)])
        phase_height = np.array([phase_height for i in range(input_size)])
        self.phase_low = nn.Parameter(torch.Tensor(phase_low))
        self.phase_height = nn.Parameter(torch.Tensor(phase_height))
        self.equal_size_conv = nn.Conv2d(1, output_size, (g+k, input_size))

    def forward(self, x):
```

Table 1: Parameter Settings for Training Speed Comparison Experiments

Func.	KAN Setting	ReLU-KAN Setting
$f_1(x) = \sin(\pi x)$	width=[1, 1], G=5, k=3	width=[1, 1], G=5, k=3
$f_2(x_1, x_2) = \sin(\pi x_1 + \pi x_2)$	width=[2, 1], G=5, k=3	width=[2, 1], G=5, k=3
$f_3(x_1, x_2) = \arctan(x_1 + x_1 x_2 + x_2^2)$	width=[2, 1, 1], G=5, k=3	width=[2, 1, 1], G=5, k=3
$f_4(x_1, x_2) = e^{\sin(\pi x_1)+x_2^2}$	width=[2, 5, 1], G=5, k=3	width=[2, 5, 1], G=5, k=3
$f_5(x_1, x_2, x_3, x_4) = e^{\sin(x_1^2+x_2^2)+\sin(x_3^2+x_4^2)}$	width=[4, 4, 2, 1], G=10, k=3	width=[4, 4, 2, 1], G=10, k=3

Table 2: Training Speed Comparison

Func.	KAN(CPU)	KAN(GPU)	ReLU-KAN(CPU)	ReLU-KAN(GPU)
f_1	2.80	6.01	0.56	0.72
f_2	3.04	7.23	0.78	0.75
f_3	5.30	12.70	1.21	1.13
f_4	11.30	23.12	1.57	1.08
f_5	19.23	34.38	2.26	1.15

```
x1 = torch.relu(x - self.phase_low)
x2 = torch.relu(self.phase_height - x)
x = x1 * x2 * self.r
x = x * x
x = x.reshape((len(x), 1, self.g, self.input_size))
x = self.equal_size_conv(x)
x = x.reshape((len(x), self.output_size))
return x
```

4 Experiments

The experimental evaluation is divided into three main parts. Firstly, we compare the training speeds of KAN and ReLU-KAN in both GPU and CPU environments. Secondly, we evaluate the fitting capabilities and convergence rates of both models under identical parameter settings. Finally, we utilize ReLU-KAN to replicate KAN's performance in the context of catastrophic forgetting.

4.1 Training Speed Comparison

We chose a function set of size 5 to compare KAN and ReLU-KAN training speeds. The parameters of KAN and ReLU-KAN are set as Table 1

The training process was conducted using the PyTorch framework. We employed the Adam optimizer for optimization and set the training set size to 1,000 samples. All models were trained for 500 iterations. The Table 2 summarizes the training times for KAN and ReLU-KAN on both GPU and CPU environments.

Based on the results presented in the Table. 2, the following conclusions can be drawn:

- **ReLU-KAN is faster than KAN:** ReLU-KAN is significantly less time-consuming than KAN in all comparisons.
- **ReLU-KAN training scales more efficiently with complexity:** As the model architecture becomes more complex, the training time increases for both KAN and ReLU-KAN. However, the improvement of time consumption of ReLU-KAN is much smaller than that of KAN
- **ReLU-KAN's GPU speed advantage grows with model complexity:** ReLU-KAN demonstrates a more significant speed advantage on GPU compared to CPU as the model complexity increases. For a single-layer model (f_1 and f_2), ReLU-KAN is 4 times faster than KAN. For a 2-layer model (f_3 and f_4), the speed difference ranges from 5 to 10 times, and for a 3-layer model(f_5), the speed difference approaches 20 times.

Table 3: Parameter Settings for Fitting Ability Comparison Experiments

Func.	KAN Setting	ReLU-KAN Setting
$f_1(x) = \sin(\pi x)$	width=[1, 1], G=5, k=3	width=[1, 1], G=5, k=3
$f_2(x) = sin(5 * \pi x) + x$	width=[2, 1], G=5, k=3	width=[2, 1], G=5, k=3
$f_3(x) = e^x$	width=[2, 1, 1], G=5, k=3	width=[2, 1, 1], G=5, k=3
$f_4(x_1, x_2) = \sin(\pi x_1 + \pi x_2)$	width=[2, 5, 1], G=5, k=3	width=[2, 5, 1], G=5, k=3
$f_5(x_1, x_2) = e^{\sin(\pi x_1)+x_2^2}$	width=[2, 5, 1], G=5, k=3	width=[2, 5, 1], G=5, k=3
$f_6(x_1, x_2, x_3, x_4) = e^{\sin(\pi x_1^2+\pi x_2^2)+\sin(\pi x_3^2+\pi x_4^2)}$	width=[4, 4, 2, 1], G=10, k=3	width=[4, 4, 2, 1], G=10, k=3

4.2 Comparison of Fitting Ability

The fitting capabilities of KAN and ReLU-KAN are then compared on three unary functions and three multivariate functions, each of which uses the parameter Settings shown in the Table 3.

To assess the performance of KAN and ReLU-KAN, we employed the Mean Squared Error (MSE) loss function as the evaluation metric and utilized the Adam optimizer for optimization. The maximum number of iterations was set to 1000.

To visualize the iterative process of both models, we plotted their loss curves. And we can visualize the fit in the following way: for univariate functions f_1, f_2, and f_3, we directly plotted their original $f(x)$ curves alongside the fitted curves, providing a clear visual representation of their fitting performance. For multivariate functions f_4, f_5, and f_6, we generated scatter plots of predicted values versus true values. The closer the scatter points lie to the line $pred = true$, the better the fitting performance.

The results in the Table. 4 indicate that ReLU-KAN exhibits a more stable training process and achieves higher fitting accuracy compared to the KAN network, given identical network structure and scale. This advantage becomes particularly pronounced for multi-layer networks, especially when fitting functions like f_2 with a higher frequency of change. In these cases, ReLU-KAN demonstrates superior fitting capabilities.

4.3 ReLU-KAN Avoids Catastrophic Forgetting

Leveraging its similar basis function structure to KAN, ReLU-KAN is expected to inherit KAN's resistance to catastrophic forgetting. To verify this, we conducted a simple experiment.

Similar to the experiment designed for KAN, the target function has five peaks. During training, the model is presented with data for only one peak at a time. The following figure illustrates the fitting curve of ReLU-KAN after each training iteration.

As shown in the Table 5, ReLU-KAN similarly has the ability to avoid catastrophic forgetting.

5 Summary and Prospect

This paper introduces a novel architecture called ReLU-KAN, which replaces the B-splines in KAN networks with a new type of basis function. Additionally, ReLU-KAN implements full matrix operations, significantly improving training speed. Experimental results demonstrate that ReLU-KAN outperforms KAN in terms of training speed, fitting capability, and stability. In future work, we aim to apply ReLU-KAN to convolutional and Transformer architectures to investigate its potential for parameter reduction without compromising model performance.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (62006110), the Natural Science Foundation of Hunan Province (2024JJ7428, 2023JJ30518) and the Scientific research project of Hunan Provincial Department of Education (22C0229) . (Corresponding author: Tao Zhu.)

References

[1] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks, 2024.

Table 4: Fitting Process and Effects

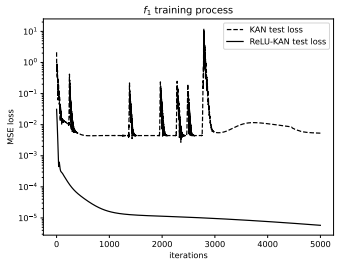
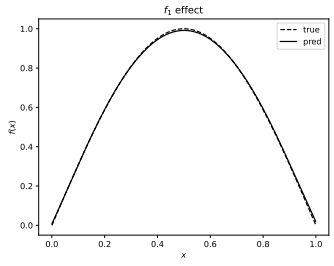
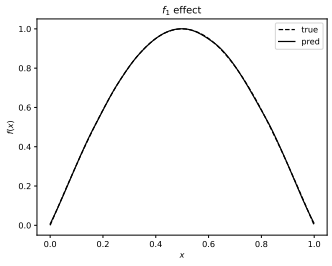
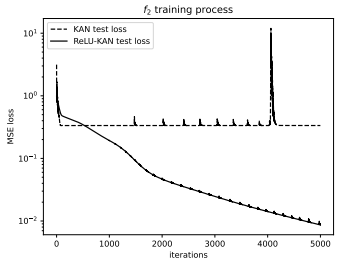
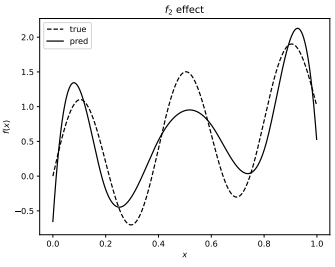
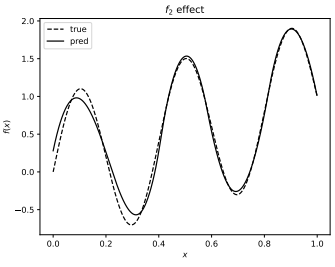
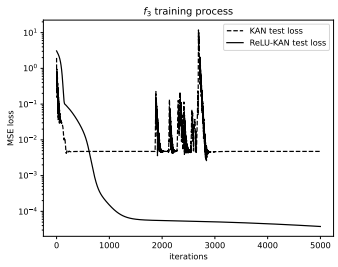
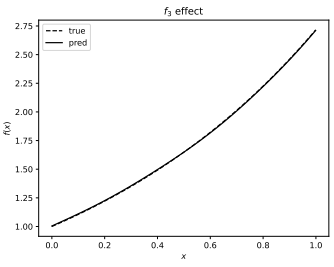
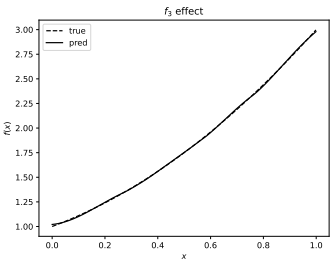
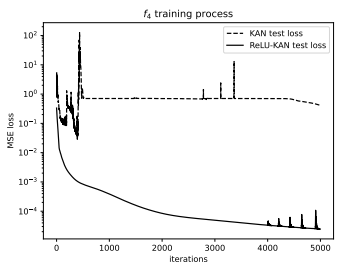
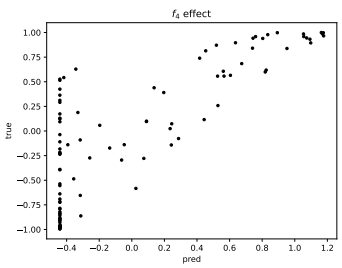
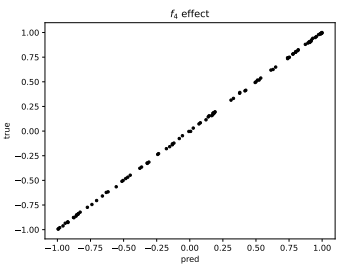
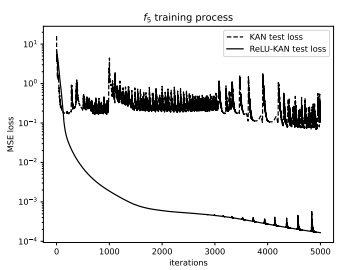
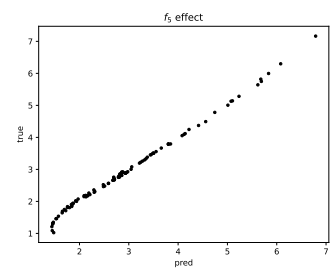
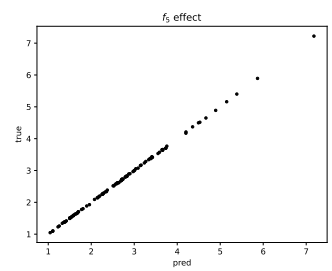
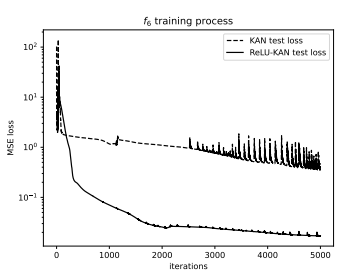
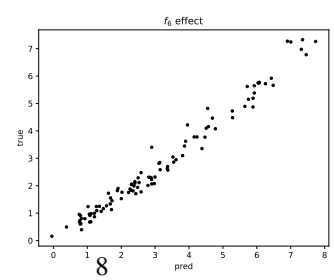
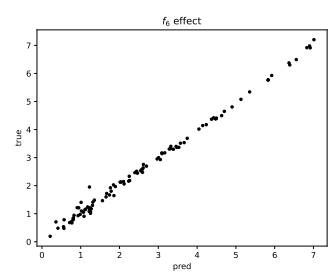
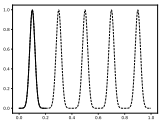
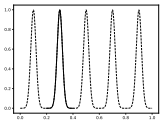
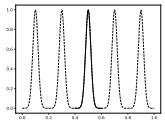
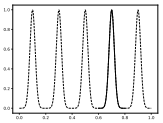
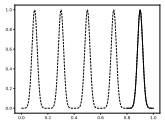
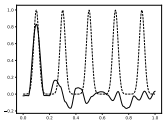
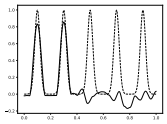
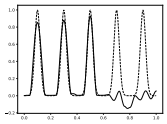
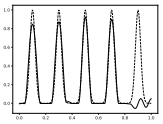
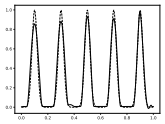
Func.	Loss curve	KAN fitted results	ReLU-KAN fitted results
f_1			
f_2			
f_3			
f_4			
f_5			
f_6			

Table 5: Catastrophic Forgetting Experiments

	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
input					
output					

[2] Diab W Abueidda, Panos Pantidis, and Mostafa E Mobasher. Deepokan: Deep operator network based on kolmogorov arnold networks for mechanics problems. *arXiv preprint arXiv:2405.19143*, 2024.

[3] Remi Genet and Hugo Inzirillo. Tkan: Temporal kolmogorov-arnold networks. *arXiv preprint arXiv:2405.07344*, 2024.

[4] Zavareh Bozorgasl and Hao Chen. Wav-kan: Wavelet kolmogorov-arnold networks. *arXiv preprint arXiv:2405.12832*, 2024.

[5] Cristian J Vaca-Rubio, Luis Blanco, Roberto Pereira, and Màrius Caus. Kolmogorov-arnold networks (kans) for time series analysis. *arXiv preprint arXiv:2405.08790*, 2024.

[6] Kazuyuki Hara, Daisuke Saito, and Hayaru Shouno. Analysis of function of rectified linear unit used in deep learning. In *2015 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2015.