Reimplementation and Analysis of Kolmogorov-Arnold Network (KAN) Architecture

Arul Bhardwaj, Medhansh Kumar, Naman Bhatia, Tanay Kulkarni, and Shrey Gupta

Birla Institute of Technology and Science, Pilani - Goa Campus, India

Abstract. Kolmogorov-Arnold Networks (KANs) are a novel class of neural networks that use B-spline based activation functions to support interpretable learning and symbolic reasoning. Inspired by the Kolmogorov-Arnold representation theorem, KANs transform each neuron's operation into a symbolic function thereby enabling the model to discover mathematical expressions from data. This research implements a modular KAN architecture comprising of spline activations, sparse transformation layers and symbolic extraction mechanisms. Using datasets like Feynman Symbolic Regression and MNIST we demonstrate the model's capacity to match or exceed traditional neural networks in tasks requiring interpretability, especially in low-data scientific domains. While the architecture enables symbolic discovery and progressive resolution through grid refinement, challenges such as overfitting, computational overhead and limited performance on complex symbolic tasks still persist. Future directions include optimizing KAN layers, enhancing symbolic regression capabilities and developing convolutional variants (ConvKAN) for structured data.

1 Introduction

Multi-Layer Perceptrons (MLPs) have long served as foundational elements in the architecture of deep learning models. MLPs are powerful tools for modelling nonlinear functions. However, they often act as black-box approximators offering limited interpretability and requiring significant parameter overhead for high-accuracy tasks. This inherent opacity and inefficiency motivate the search for more interpretable and data-efficient alternatives.

Kolmogorov-Arnold Networks (KANs) present a new paradigm inspired by the Kolmogorov-Arnold representation theorem, which says that any multivariate continuous function can be expressed as a finite sum of univariate functions. In contrast to MLPs where non-linearities are applied at nodes, KANs shift the non-linearity to the edges by replacing weight parameters with learnable univariate functions parametrized via B-splines. This architectural shift allows KANs to model symbolic expressions, enhancing interpretability while preserving and often exceeding the expressive power of traditional networks.

In this work, we implement a modular and interpretable version of KANs using Python and PyTorch. Our architecture includes spline-based activation functions, sparse transformation layers and symbolic extraction methods with progressive grid refinement to improve accuracy during training. Symbolic formulas are discovered using the SymPy library and regularization techniques are employed to encourage sparsity and simplify the model.

We evaluate our implementation on tasks ranging from symbolic regression using the Feynman dataset to image classification on MNIST. Our results confirm the strength of KANs in low-data and scientific settings, where symbolic reasoning and interpretability are crucial. However, we also observe challenges such as overfitting, training instability, and increased computational cost highlighting the need for further optimizations in spline handling and architecture design.

Through this exploration we aim to contribute a reproducible and extensible framework for KANs while empirically validating their potential as interpretable and accurate alternatives to MLPs in real-world learning tasks.

1.1 Foundation

Building upon this foundation, it is essential to understand the theoretical motivation that distinguishes KANs from traditional neural architectures. The **Kolmogorov–Arnold representation theorem**, formulated by Andrey Kolmogorov and Vladimir Arnold asserts that any multivariate continuous function $f : [0, 1]^n \to R$ can be expressed in the form:

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$$

where $\phi_{q,p}$ and Φ_q are univariate continuous functions. This elegant formulation implies that the complexity of multivariate functions can be decomposed into compositions of simpler one-dimensional functions, making the representation not only powerful but also potentially interpretable. While the original formulation of the theorem was regarded as theoretically profound but practically impractical due to the possible non-smoothness and fractal nature of the component functions, modern techniques like B-spline parameterization breathe new life into its applicability in machine learning.

KANs leverage this theorem by replacing the traditional fixed activation functions of MLPs with **trainable univariate functions on edges**, each represented using B-splines. These splines provide smooth and flexible local approximations, and their resolutions can be progressively refined during training. Moreover, this edge-centric formulation aligns naturally with symbolic expression discovery, allowing the network to not only fit data but also *uncover underlying functional relationships*.

Our implementation incorporates many of the practical enhancements proposed in recent KAN literature, including grid refinement strategies, L1 and entropy-based sparsification for pruning and symbolic extraction interfaces that convert learned representations into interpretable formulas. These features make KANs suitable for domains where understanding the learned model is as important as its predictive accuracy such as physics, engineering and mathematical modeling.

Despite their promise, KANs are not without limitations. The computational overhead of learning B-spline activations, coupled with a high sensitivity to initialization and overfitting poses challenges for scalability and general-purpose use. Furthermore, symbolic regression on high-dimensional or noisy functions remains a bottleneck, particularly where compositional structure is weak or ambiguous.

Nonetheless, the ability of KANs to balance accuracy, interpretability and symbolic representation offers a compelling direction for neural network research. As we continue to refine their implementation and explore new applications, KANs may serve not just as alternatives to MLPs but as a bridge between black-box deep learning and white-box symbolic reasoning.

Kolmogorov–Arnold Networks (KAN)

Multi-Layer Perceptrons (MLPs) are motivated by the universal approximation theorem. In contrast, the Kolmogorov–Arnold representation theorem inspires a new class of neural networks called Kolmogorov–Arnold Networks (KAN). In the following we describe the theoretical background, the network architecture, and the approximation guarantees of KANs.

2 Kolmogorov–Arnold Representation Theorem

Vladimir Arnold and Andrey Kolmogorov established that if a function f is continuous and multivariate on a bounded domain, then it can be written as a finite composition of univariate continuous functions and the binary operation of addition. More specifically, for a smooth function

$$f : [0,1]^n \to R,$$

one may represent

$$f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right),$$

where each $\phi_{q,p} : [0,1] \to R$ is a univariate function and $\Phi_q : R \to R$. In essence, this theorem indicates that addition is the only fundamental multivariate operation since every function can be composed entirely by univariate functions and summation.

Although this result suggests that learning a high-dimensional function reduces to learning a polynomial number of one-dimensional functions, the potential non-smoothness (or even fractal behaviour) of these univariate components renders the practical approach challenging. Many functions encountered in scientific and real-world applications, however, are smooth and exhibit sparse compositional structures, which motivates the use of KANs.

3 KAN Architecture

Consider a supervised learning task with input-output pairs $\{(x^{(i)}, y^{(i)})\}$ and the goal of finding a function f such that $y^{(i)} \approx f(x^{(i)})$ for all data points. The Kolmogorov–Arnold theorem implies that it suffices to learn appropriate univariate functions $\phi_{q,p}$ and Φ_q.

Since all functions to be learned are one-dimensional, each can be parameterized using B-spline curves with learnable coefficients corresponding to local B-spline basis functions. The prototype KAN corresponds to a two-layer network in which the activation functions are applied on the edges (instead of at the nodes) and the hidden layer has width $2n + 1$ for an input of dimensionality n.

To generalize to deeper and wider architectures, a *KAN layer* with n_{in}-dimensional inputs and n_{out}-dimensional outputs is defined as a matrix of one-dimensional functions:

$$\Phi = \Big\{\phi_{q,p} : p = 1, 2, \ldots, n_{in};\ q = 1, 2, \ldots, n_{out}\Big\}.$$

In the classical representation, the inner functions form a KAN layer with $n_{in} = n$ and $n_{out} = 2n + 1$, while the outer functions form a KAN layer with $n_{in} = 2n + 1$ and $n_{out} = 1$. Hence, the standard Kolmogorov–Arnold representation is a composition of two KAN layers.

For a deeper network, let the network shape be represented by the integer array

$$[n_0, n_1, \ldots, n_L],$$

where n_i denotes the number of nodes in the ith layer. Denote the ith neuron in layer l by (l, i) with activation $x_{l,i}$. The activation function connecting neuron (l, i) to neuron $(l + 1, j)$ is denoted by $\phi_{l,j,i}$. Its pre-activation is $x_{l,i}$, and the post-activation is defined as

$$\tilde{x}_{l,j,i}\phi_{l,j,i}(x_{l,i}).$$

Thus, the activation of neuron $(l + 1, j)$ is computed as

$$x_{l+1,j} = \sum_{i=1}^{n_l} \tilde{x}_{l,j,i} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}).$$

In matrix form, this is written as

$$x_{l+1} = \Phi_l\, x_l,$$

where Φ_l is the function matrix for layer l.

A general KAN network with L layers is then given by the composition

$$\mathrm{KAN}(x) = \Big(\Phi_{L-1} \circ \Phi_{L-2} \circ \cdots \circ \Phi_0\Big)(x).$$

When the output is scalar (i.e., $n_L = 1$), this can be written as a nested sum:

$$f(x) = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1,1,i_{L-1}}\left(\sum_{i_{L-2}=1}^{n_{L-2}} \cdots \sum_{i_1=1}^{n_1} \phi_{0,i_1,i_0}(x_{i_0}) \cdots\right).$$

For comparison, a standard MLP is typically expressed as

$$\mathrm{MLP}(x) = \Big(W_{L-1} \circ \sigma \circ W_{L-2} \circ \sigma \circ \cdots \circ W_0\Big)(x),$$

where alternating linear transformations W_i and nonlinearities σ are used. In contrast, KANs combine these operations within unified function matrices Φ.

Implementation Details

Several key techniques are employed to ensure that KANs are optimizable:

– **Residual Activation Functions:** Each activation function is expressed as the sum of a basis function $b(x)$ and a spline function:

$$\phi(x) = w_b\, b(x) + w_s\, spline(x).$$

Here,

$$b(x) = silu(x) = \frac{x}{1 + e^{-x}},$$

and the spline function is defined as

$$spline(x) = \sum_i c_i\, B_i(x),$$

with c_i being trainable coefficients.

– **Initialization Scales:** The spline part is initialized with $w_s = 1$ and $spline(x) \approx 0$, while the coefficient w_b is initialized using Xavier initialization.
– **Adaptive Spline Grids:** Each spline grid is updated on the fly based on the input activations. This ensures that the activation functions remain well-behaved even when inputs move outside their initial bounds.

Assume a network of depth L with equal layer widths $n_0 = n_1 = \cdots = n_L = N$ and each spline has order k (usually $k = 3$) over G intervals (i.e., $G + 1$ grid points). Then the total number of parameters is on the order of

$$O\Big(N^{2L}(G + k)\Big) \sim O\Big(N^{2L}G\Big).$$

In contrast, an MLP of depth L and width N requires $O\big(N^{2L}\big)$ parameters. In practice, KANs can use significantly smaller values of N than MLPs, leading to a reduction in parameters along with improved generalization and interpretability. Notably, for one-dimensional problems, setting $N = L = 1$ reduces the KAN to a simple spline approximation.

4 KAN's Approximation Abilities and Scaling Laws

While the two-layer Kolmogorov–Arnold representation (with network shape $[n, 2n+1, 1]$) may produce non-smooth functions, deeper KAN representations can yield smoother activations. For example, the function

$$f(x_1, x_2, x_3, x_4) = \exp\Big(\sin(x_1^2 + x_2^2)\Big) + \sin(x_3^2 + x_4^2)$$

admits a smooth representation using a three-layer KAN even if a two-layer version with smooth activations may not exist.

To facilitate approximation analysis, consider the representation

$$f(x) = \Big(\Phi_{L-1} \circ \Phi_{L-2} \circ \cdots \circ \Phi_0\Big)(x),$$

where each function $\phi_{l,j,i}$ is assumed to be $(k + 1)$-times continuously differentiable. To emphasize the dependence on the discretization grid, denote the B-spline approximations on a grid of size G as $\Phi^G_{l,i,j}$.

Theorem 1 (Approximation Theory, KAT). *Let $x = (x_1, x_2, \ldots, x_n)$ and suppose that a function f admits the representation above with every function $\phi_{l,j,i}$ being $(k+1)$-times continuously differentiable. Then there exists a constant C (depending on f and its representation) such that for any $0 \le m \le k$,*

$$\Big\|f - \Big(\Phi^G_{L-1} \circ \Phi^G_{L-2} \circ \cdots \circ \Phi^G_0\Big)(x)\Big\|_{C^m} \le C\, G^{-k-1+m},$$

where the C^m-norm is defined by

$$\|g\|_{C^m} = \max_{|\beta| \le m} \sup_{x \in [0,1]^n} \Big|D^\beta g(x)\Big|.$$

Proof. The proof follows from classical one-dimensional B-spline approximation theory and the boundedness of the functions $\phi_{l,j,i}$ on a bounded domain. For each $\phi_{l,j,i}$, there exist B-spline approximations $\Phi^G_{l,i,j}$ satisfying

$$\Big\|\phi_{l,j,i} - \Phi^G_{l,i,j}\Big\|_{C^m} \le C\, G^{-k-1+m},$$

for any $0 \le m \le k$. By propagating this error through the composition of layers, the overall error bound for f is obtained.

This theorem implies that, under the smoothness assumptions, KANs with a finite grid size can approximate functions with an error rate independent of the input dimension, thereby alleviating the curse of dimensionality. Taking $m = 0$ recovers the L^∞ norm accuracy, which in turn provides a bound on the RMSE over the finite domain.

Neural Scaling Laws

Neural scaling laws empirically show that the test loss—typically measured as the RMSE ℓ—decreases with the number of model parameters N as

$$\ell \propto N^{-\alpha},$$

where α is the scaling exponent. Several theoretical approaches predict different values for α:

- **Sharma and Kaplan:** When the model consists of piecewise polynomials of order k and the data lies on a d-dimensional manifold, then $\alpha = \frac{k+1}{d}$.
- **Michaud et al.:** By considering compositional structures with unary and binary operations, one derives $\alpha = \frac{k+1}{2}$.
- **Poggio et al.:** For function classes where derivatives are continuous up to order m, one requires $N = O\big(\epsilon^{-2m}\big)$ parameters to achieve error ϵ, corresponding to $\alpha = \frac{m}{2}$.

Under the assumption of smooth Kolmogorov–Arnold representations—where the high-dimensional function is decomposed into several one-dimensional functions—the network achieves

$$\alpha = k + 1.$$

For instance, using cubic splines (i.e., $k = 3$) yields $\alpha = 4$, a notably high scaling exponent compared to other approaches.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Displayed equations are centered and set on a separate line.

$$x + y = z \tag{1}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead.

5 Results

In this section, we present the empirical performance of our Kolmogorov–Arnold Network (KAN) implementation on the Feynman datasets and comparing them with the original paper.

Table 1: Performance of the original KAN architectures and MLP across selected Feynman equations

Feynman Eq.	Human KAN	Unpruned KAN	Pruned KAN	Paper KAN RMSE	Unpruned KAN RMSE	Pruned KAN RMSE	MLP RMSE
I.6.2	[2,3,3,3,1]	[2,3,3,3,1]	[2,3,3,1]	0.018238	0.011037	0.010432	0.015586
I.9.18	[6,3,3,3,1]	[6,3,3,3,1]	[5,3,3,2,1]	0.050456	0.049958	0.050025	0.049996
I.15.3x	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.004421	0.003968	0.000204	0.002010
II.2.6.15a	[3,3,3,3,3,1]	[3,3,3,3,3,1]	[3,3,3,1,1,1]	0.020239	0.025122	0.018975	0.017897
II.11.7	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,1]	0.035665	0.036790	0.034850	0.035010
II.36.38	[3,4,1]	[3,4,1]	[3,3,1]	0.010642	0.002480	0.000236	0.000093
II.11.27	[2,3,3,3,3,1]	[2,3,3,3,3,1]	[2,3,2,2,3,1]	0.002980	0.000238	0.000165	0.000226
III.9.52	[3,3,3,3,3,1]	[3,3,3,3,3,1]	[3,3,2,1]	0.068071	0.068046	0.068181	0.068053
I.30.3	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,3,1]	0.008827	0.011068	0.007223	0.007427
I.37.4	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,3,1]	0.004606	0.000848	0.000437	0.000692
I.40.1	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.111538	0.093825	0.070006	0.114678
I.50.26	[4,3,3,3,1]	[4,3,3,3,1]	[4,3,3,3,1]	0.147042	0.125610	0.104963	0.146903
II.2.42	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.032351	0.032236	0.032371	0.021441

6 Simplification techniques

6.1 Sparsification

For MLPs, L1 regularization of linear weights is used to favor sparsity. KANs can adapt this high-level idea, but need two modifications:

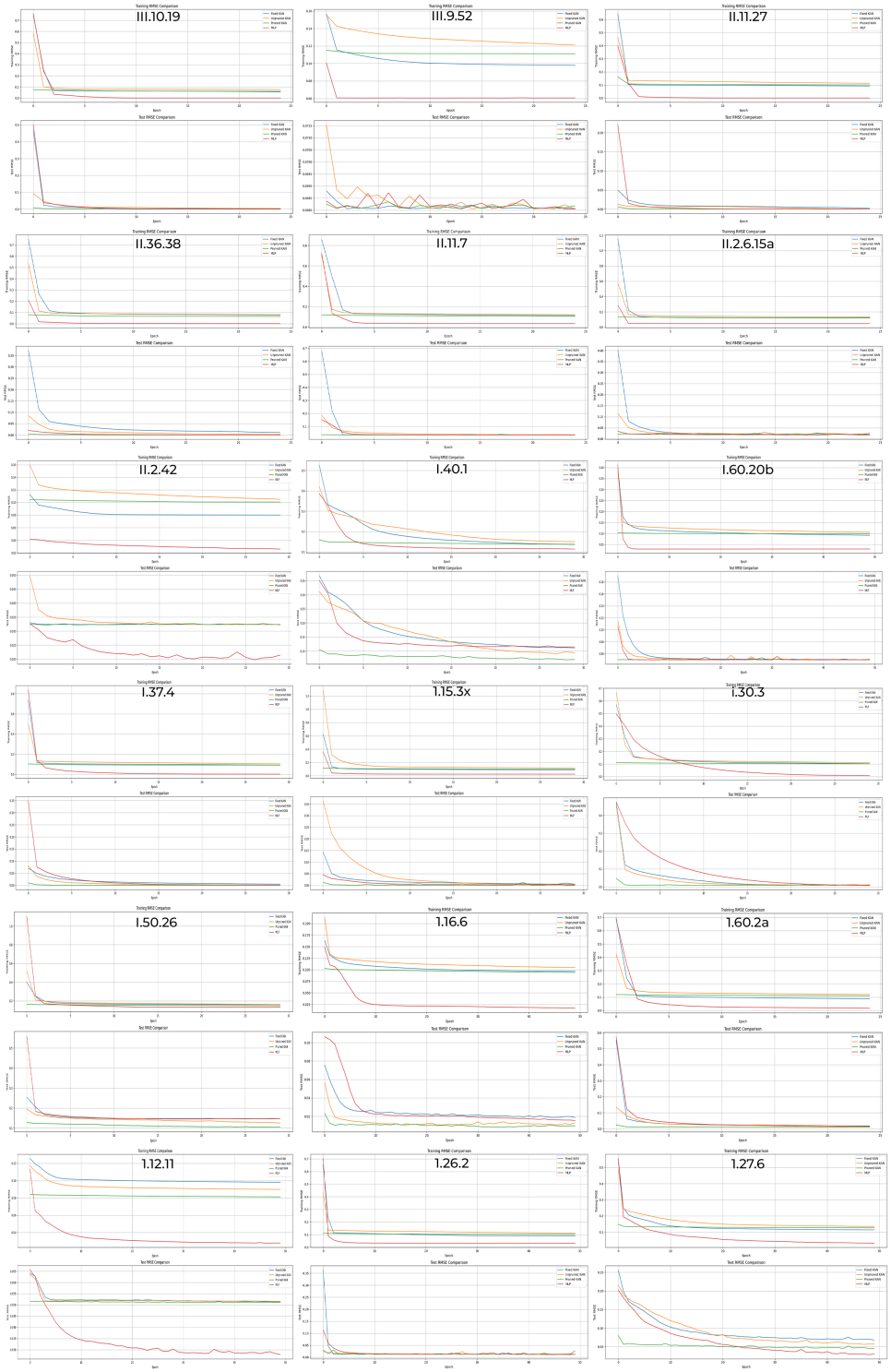

Fig. 1: Empirical performance of our KAN implementation.

Table 2: Performance comparison of our KAN architecture and MLP

Equation	Human KAN	Unpruned KAN	Pruned KAN	Fixed KAN	Unpruned KAN RMSE	Pruned KAN RMSE	MLP RMSE
I.6.2	[2,3,3,3,1]	[2,3,3,3,1]	[2,3,3,1]	0.0182	0.0110	0.0104	0.0156
I.9.18	[6,3,3,3,1]	[6,3,3,3,1]	[5,3,3,2,1]	0.0505	0.0500	0.0500	0.0500
I.15.3x	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.0044	0.0040	0.0002	0.0020
II.2.6.15a	[3,3,3,3,3,1]	[3,3,3,3,3,1]	[3,3,3,1,1,1]	0.0202	0.0251	0.0190	0.0179
II.11.7	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,1]	0.0357	0.0368	0.0349	0.0350
II.36.38	[3,4,1]	[3,4,1]	[3,3,1]	0.0106	0.0025	0.0002	0.0001
II.11.27	[2,3,3,3,3,1]	[2,3,3,3,3,1]	[2,3,2,2,3,1]	0.0030	0.0002	0.0002	0.0002
III.9.52	[3,3,3,3,3,1]	[3,3,3,3,3,1]	[3,3,2,1]	0.0681	0.0680	0.0682	0.0681
I.30.3	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,3,1]	0.0088	0.0111	0.0072	0.0074
I.37.4	[3,3,3,3,1]	[3,3,3,3,1]	[3,3,3,3,1]	0.0046	0.0008	0.0004	0.0007
I.40.1	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.1115	0.0938	0.0700	0.1147
I.50.26	[4,3,3,3,1]	[4,3,3,3,1]	[4,3,3,3,1]	0.1470	0.1256	0.1050	0.1469
II.2.42	[5,3,3,3,1]	[5,3,3,3,1]	[5,3,3,3,1]	0.0324	0.0322	0.0324	0.0214

1. There is no linear "weight" in KANs. Linear weights are replaced by learnable activation functions, so we should define the L1 norm of these activation functions.
2. We find L1 to be insufficient for sparsification of KANs; instead an additional entropy regularization is necessary (see Appendix C for more details).

We define the L1 norm of an activation function ϕ to be its average magnitude over its N_p inputs, i.e.,

$$|\phi|_1 \equiv \frac{1}{N_p}\sum_{s=1}^{N_p}|\phi(x^{(s)})|.2.17 \tag{2}$$

Then for a KAN layer Φ with n_{in} inputs and n_{out} outputs, we define the L1 norm of Φ to be the sum of L1 norms of all activation functions, i.e.,

$$|\Phi|_1 \equiv \sum_{i=1}^{n_{in}}\sum_{j=1}^{n_{out}}|\phi_{i,j}|_1.2.18 \tag{3}$$

In addition, we define the entropy of Φ to be

$$S(\Phi) \equiv -\sum_{i=1}^{n_{in}}\sum_{j=1}^{n_{out}}\frac{|\phi_{i,j}|_1}{|\Phi|_1}\log\left(\frac{|\phi_{i,j}|_1}{|\Phi|_1}\right).2.19 \tag{4}$$

The total training objective ℓ_{total} is the prediction loss ℓ_{pred} plus L1 and entropy regularization of all KAN layers:

$$\ell_{total} = \ell_{pred} + \lambda\left(\mu_1\sum_{l=0}^{L-1}|\Phi_l|_1 + \mu_2\sum_{l=0}^{L-1}S(\Phi_l)\right), 2.20 \tag{5}$$

where μ_1, μ_2 are relative magnitudes usually set to $\mu_1 = \mu_2 = 1$, and λ controls overall regularization magnitude.

6.2 Visualization

When we visualize a KAN, to get a sense of magnitudes, we set the transparency of an activation function $\phi_{l,i,j}$ proportional to $\tanh(\beta A_{l,i,j})$ where $\beta = 3$. Hence, functions with small magnitude appear faded out to allow us to focus on important ones.

6.3 Pruning

After training with sparsification penalty, we may also want to prune the network to a smaller subnetwork. We sparsify KANs on the node level (rather than on the edge level). For each node (say the ith neuron in the lth layer), we define its incoming and outgoing score as

$$I_{l,i} = \max_k(|\phi_{l-1,i,k}|_1), \quad O_{l,i} = \max_j(|\phi_{l+1,j,i}|_1), 2.21 \tag{6}$$

and consider a node to be important if both incoming and outgoing scores are greater than a threshold hyperparameter $\theta = 10^{-2}$ by default. All unimportant neurons are pruned.

6.4 Symbolification

In cases where we suspect that some activation functions are in fact symbolic (e.g., cos or log), we provide an interface to set them to be a specified symbolic form, `fix_symbolic(l,i,j,f)` can set the (l, i, j) activation to be f. However, we cannot simply set the activation function to be the exact symbolic formula, since its inputs and outputs may have shifts and scalings. So, we obtain preactivations x and postactivations y from samples, and fit affine parameters (a, b, c, d) such that $y \approx cf(ax + b) + d$. The fitting is done by iterative grid search of a, b and linear regression.

Besides these techniques, we provide additional tools that allow users to apply more fine-grained control to KANs, listed in Appendix A.

7 A toy example: how humans can interact with KANs

Above we have proposed a number of simplification techniques for KANs. We can view these simplification choices as buttons one can click on. A user interacting with these buttons can decide which button is most promising to click next to make KANs more interpretable. We use an example below to showcase how a user could interact with a KAN to obtain maximally interpretable results.

Let us again consider the regression task

$$f(x, y) = \exp \sin(\pi x) + y^2. \tag{7}$$

Given data points (x_i, y_i, f_i), $i = 1, 2, \ldots, N_p$, a hypothetical user Alice is interested in figuring out the symbolic formula. The steps of Alice's interaction with the KANs are described below (illustrated in Figure **??**):

Step 1: Training with sparsification. Starting from a fully-connected [2, 5, 1] KAN, training with sparsification regularization can make it quite sparse. 4 out of 5 neurons in the hidden layer appear useless, hence we want to prune them away.

Step 2: Pruning. Automatic pruning is seen to discard all hidden neurons except the last one, leaving a [2, 1, 1] KAN. The activation functions appear to be known symbolic functions.

Step 3: Setting symbolic functions. Assuming that the user can correctly guess these symbolic formulas from staring at the KAN plot, they can set `fix_symbolic(0,0,0,'sin')`
`fix_symbolic(0,1,0,'x^2')`
`fix_symbolic(1,0,0,'exp')`. 2.23

In case the user has no domain knowledge or no idea which symbolic functions these activation functions might be, we provide a function `suggest_symbolic` to suggest symbolic candidates.

Step 4: Further training. After symbolifying all the activation functions in the network, the only remaining parameters are the affine parameters. We continue training these affine parameters, and when we see the loss dropping to machine precision, we know that we have found the correct symbolic expression.

Step 5: Output the symbolic formula. Sympy is used to compute the symbolic formula of the output node. The user obtains $1.0e^{1.0y^2+1.0\sin(3.14x)}$, which is the true answer (we only displayed two decimals for π).

Remark: Why not symbolic regression (SR)? It is reasonable to use symbolic regression for this example. However, symbolic regression methods are in general brittle and hard to debug. They either return a success or a failure in the end without outputting interpretable intermediate results. In contrast, KANs do continuous search (with gradient descent) in function space, so their results are more continuous and hence more robust. Moreover, users have more control over KANs as compared to SR due to KANs' transparency. The way we visualize KANs is like displaying KANs' "brain" to users, and users can perform "surgery" (debugging) on KANs. This level of control is typically unavailable for SR. We will show examples of this in Section 4.4. More generally, when the target function is not symbolic, symbolic regression will fail but KANs can still provide something meaningful. For example, a special function (e.g., a Bessel function) is impossible to SR to learn unless it is provided in advance, but KANs can use splines to approximate it numerically anyway

8 Detailed Implementation of Symbolic Discovery

The key advantage of KANs is their interpretability through symbolic discovery. Based on our implementation, we explain how KAN enables symbolic representations of functions.

8.1 Implementation Workflow for Symbolic Discovery

We present a structured approach to discovering symbolic formulas from data, which we have implemented in practice.

Step 1: Training with Sparsification We begin with an overparameterized KAN (typically [2,5,1] for 2D input functions) and apply regularization:

```
[caption={KAN training with sparsification}]
model.fit(
    dataset,
    opt="LBFGS",
    steps=100,
    lamb=0.01,              # Regularization weight
    lamb_l1=1.0,            # L1 regularization for sparsity
    lamb_entropy=2.0,       # Entropy regularization
    reg_metric='edge_forward_spline_n'
)
```

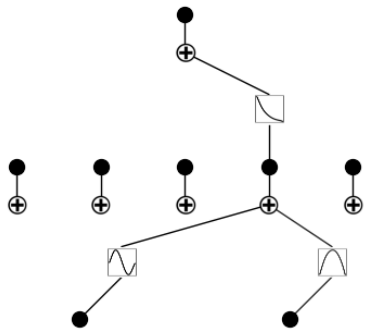

Fig. 2: Sparsed Network.

Step 2: Pruning After sparsification, KAN prunes unnecessary neurons and edges:

```
[caption={KAN pruning step}]
model.get_act(dataset['train_input'])
model.attribute()

# Remove weak nodes and edges
pruned_model = model.prune(node_th=1e-2, edge_th=3e-2)
```

For example, pruning a model trained on $f(x, y) = \exp(\sin(\pi x) + y^2)$ reduces it from [2,5,1] to [2,1,1].

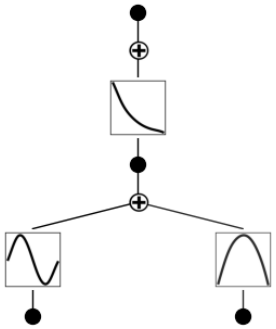

Fig. 3: Pruned Network.

Step 3: Detecting Symbolic Functions Symbolic functions are automatically detected and selected based on fit quality:

```
[caption={Automatic symbolic function selection}]
def auto_select_symbolic_functions(model, dataset, threshold=0.9):
    selected_functions = {}
    for l in range(len(model.width_in) - 1):
        for i in range(model.width_in[l]):
            for j in range(model.width_out[l + 1]):
                if model.act_fun[l].mask[i, j] == 0:
                    continue
                name, fun, r2, c = model.suggest_symbolic(l, i, j)
                if r2 > threshold:
                    selected_functions[(l, i, j)] = name
    return selected_functions
```

KAN identifies matches like:

- sin for (layer 0, input 0, output 0)
- x^2 for (layer 0, input 1, output 0)
- exp for (layer 1, input 0, output 0)

These are then applied with affine transformation:

```
[caption={Apply symbolic functions with affine parameters}]
for (l, i, j), func_name in selected_functions.items():
    pruned_model.fix_symbolic(l, i, j, func_name)
    # Internally fits: y     c * f(a * x + b) + d
```

Step 4: Robust Training of Affine Parameters We fine-tune the symbolic components in a staged manner:

```
[caption={Fine-tuning affine parameters}]
pruned_model.affine_trainable = True

for i, steps in enumerate([30, 50, 100]):
    pruned_model.fit(
        dataset,
```

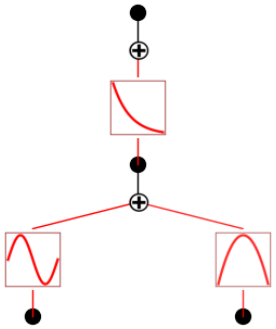

Fig. 4: Symbolics Added.

```
        opt="LBFGS",
        steps=steps,
        lamb=0.0
    )
```

Step 5: Output the Symbolic Formula Once fine-tuning is done, we extract the final expression:

```
[caption={Extracting the symbolic formula}]
formula, vars = pruned_model.symbolic_formula()
```

For $f(x, y) = \exp(\sin(\pi x) + y^2)$, the output is:

$$1.0 \cdot \exp(1.0 \cdot y^2 + 1.0 \cdot \sin(3.14 \cdot x))$$

8.2 Key Implementation Insights

Visualization Aids Interpretation To interpret the internal splines, we use the following plotting utility:

```
[caption={Visualizing learned spline functions}]
def visualize_spline(model, layer, input_idx, output_idx, title=None,
    save_path=None):
    x = torch.linspace(-1, 1, 100).unsqueeze(1)
    kan_layer = model.act_fun[layer]
    with torch.no_grad():
        y_values = [...]  # Compute activation values here
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.plot(x.numpy(), y_values)
    ax.set_xlabel('Input')
    ax.set_ylabel('Output')
    ax.set_title(title or f'Spline: Layer {layer}, Input {input_idx},
        Output {output_idx}')
```

Symbolic Function Suggestion Engine Symbolic matches are discovered via regression with R^2 scoring:

```
[caption={Sketch of suggest\_symbolic method}]
def suggest_symbolic(self, layer, input_idx, output_idx, verbose=False):
    # 1. Sample input points
    # 2. Evaluate spline output
    # 3. Fit candidate symbolic forms (sin, cos, x^2, log, etc.)
    # 4. Return best match with R   score
```

Progressive Optimization Gradual training improves final accuracy:

```
[caption={Progressive training loop}]
training_log = []

for i, steps in enumerate([30, 50, 100]):
    pruned_model.fit(dataset, opt="LBFGS", steps=steps, lamb=0.0)
    with torch.no_grad():
        preds = pruned_model(dataset['test_input'])
        mse = torch.mean((preds - dataset['test_label']) ** 2)
        training_log.append((i+1, steps, mse.item()))
```

Symbolic Formula Validation We validate the formula over a grid of test points:

```
[caption={Validation of symbolic discovery}]
with open("comparison_data.txt", "w") as f:
    x = torch.linspace(-1, 1, 10)
    y = torch.linspace(-1, 1, 10)
    xx, yy = torch.meshgrid(x, y, indexing='ij')
    points = torch.stack([xx.flatten(), yy.flatten()], dim=1)

    model_preds = pruned_model(points)
    true_preds = target_function(points)
    diff = torch.abs(model_preds - true_preds)
```

8.3 Advantages Over Traditional Symbolic Regression

Our implementation confirms the advantages of KANs over traditional symbolic regression methods:

1. **Transparency**: KANs provide visibility into the symbolic discovery process, allowing users to see the structure of the learned function.
2. **Continuous optimization**: By using gradient-based optimization, KANs provide a smooth path to finding symbolic expressions rather than the discrete search used in traditional symbolic regression.
3. **Interactive refinement**: Users can interactively modify and experiment with different symbolic forms, allowing for domain knowledge incorporation.
4. **Robustness**: Even when exact symbolic forms aren't found, KANs can provide useful approximations with splines.

9 Future Works

While our implementation of Kolmogorov-Arnold Networks demonstrates promising results in terms of interpretability and accuracy, several challenges and open directions remain for future exploration:

1. **Improved Symbolic Regression on Complex Functions:** KANs show strong performance on smooth and compositional functions, but struggle with more intricate symbolic structures, such as

those involving discontinuities, piecewise definitions, or higher-order partial differential equations (PDEs). Developing more robust symbolic discovery algorithms or hybridizing with symbolic regression engines could help address these limitations.

2. **Optimization of B-Spline Activations:** Training spline-based activations can be computationally expensive and sensitive to initialization. Future work could explore better initialization schemes, gradient clipping strategies, or even alternative basis functions (e.g., wavelets or Fourier bases) that offer improved convergence properties.
3. **Dynamic Grid Management:** While our implementation uses progressive grid refinement, this is performed uniformly and heuristically. A more adaptive, data-driven grid refinement mechanism—possibly guided by curvature or error feedback—could enhance both training speed and approximation quality.
4. **ConvKAN and Structured Data Extensions:** Extending the KAN architecture to structured domains such as images and graphs remains largely unexplored. A promising direction is the development of *ConvKANs*, which incorporate convolutional priors alongside spline-based symbolic reasoning for spatial data.
5. **Scalability to High Dimensions:** As the number of input dimensions increases, the number of spline activations and their associated parameters also grows quadratically. Techniques such as low-rank approximations, sparse connections, or attention-inspired mechanisms could be used to mitigate this growth.
6. **Symbolic Pruning and Architecture Discovery:** Our pruning strategy relies on manual thresholds and regularization. Future efforts could incorporate neural architecture search (NAS) or reinforcement learning to automatically infer minimal symbolic structures from data with minimal supervision.
7. **Integration with Domain Knowledge:** In scientific domains, it is often known that certain operations (e.g., sin, exp, log) are likely components of the target function. Leveraging such priors directly during training—either through activation regularization or symbolic constraints—could accelerate convergence and improve symbolic fidelity.
8. **Unified Interpretability Interface:** To make KANs truly user-friendly for scientific and industrial users, future work could focus on developing a visual, interactive interface that allows users to observe symbolic formulas, prune neurons, and suggest symbolic replacements in real time.

These directions not only aim to strengthen KANs as a research tool but also move them closer to being viable, interpretable components in real-world scientific AI systems.These directions not only aim to strengthen KANs as a research tool but also move them closer to being viable, interpretable components in real-world scientific AI systems.

References

1. R. Qiu, Y. Miao, S. Wang, L. Yu, Y. Zhu, and X.-S. Gao, "PowerMLP: An Efficient Version of KAN," *arXiv preprint arXiv:2412.13571*, 2024.
2. Q. Qiu, T. Zhu, H. Gong, L. Chen, and H. Ning, "ReLU-KAN: New Kolmogorov-Arnold Networks that Only Need Matrix Addition, Dot Multiplication, and ReLU," *arXiv preprint arXiv:2406.02075*, 2024.
3. S. Somvanshi, S. A. Javed, M. M. Islam, D. Pandit, and S. Das, "A Survey on Kolmogorov-Arnold Network," *arXiv preprint arXiv:2411.06078*, 2024.
4. B. C. Koenig, S. Kim, and S. Deng, "LeanKAN: A Parameter-Lean Kolmogorov-Arnold Network Layer with Improved Memory Efficiency and Convergence Behavior," *arXiv preprint arXiv:2502.17844*, 2025.
5. M. G. Altarabichi, "Rethinking the Function of Neurons in KANs," *arXiv preprint arXiv:2407.20667*, 2024.
6. H. Guo, F. Li, J. Li, and H. Liu, "KAN v.s. MLP for Offline Reinforcement Learning," *arXiv preprint arXiv:2409.09653*, 2024.
7. Y. Wang, J. Sun, J. Bai, C. Anitescu, M. S. Eshaghi, X. Zhuang, T. Rabczuk, and Y. Liu, "Kolmogorov Arnold Informed Neural Network: A Physics-Informed Deep Learning Framework for Solving Forward and Inverse Problems Based on Kolmogorov Arnold Networks," *arXiv preprint arXiv:2406.11045*, 2024.
8. K. Mohan, H. Wang, and X. Zhu, "KANs for Computer Vision: An Experimental Study," *arXiv preprint arXiv:2411.18224*, 2024.
9. Y. Wang, J. W. Siegel, Z. Liu, and T. Y. Hou, "On the Expressiveness and Spectral Bias of KANs," *arXiv preprint arXiv:2410.01803*, 2024.
10. I. Drokin, "Kolmogorov-Arnold Convolutions: Design Principles and Empirical Studies," *arXiv preprint arXiv:2407.01092*, 2024.