

KANs and DeepOKANs

Brief Introduction

MAS

IISc, Bengaluru

August 25, 2024



① Introduction to Data Fitting

② Introduction to KAN

③ KAT

④ Results

- 1 Introduction to Data Fitting
- 2 Introduction to KAN
- 3 KAT
- 4 Results

Representing Smooth Functions

Why do we need activation functions?

After each Linear layer, we usually apply a nonlinear activation function. Why?

$$o_1 = xw_1^T + b_1$$

$$o_2 = (o_1)w_2^T + b_2$$

$$o_2 = (xw_1^T + b_1)w_2^T + b_2$$

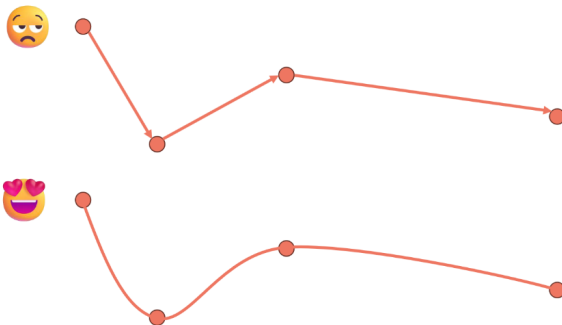
$$o_2 = xw_1^Tw_2^T + b_1w_2^T + b_2$$

As you can see, if we do not apply any activation functions, the output will just be a linear combination of the inputs, which means that our MLP will not be able to learn any non-linear mapping between the input and output, which represents most of the real-world data.

Representing Smooth Functions

Introduction to data fitting

Imagine you're making a 2D game and you want animate your sprite (character) to pass through a series of points. One way would be to make a straight line from one point to the next, but that wouldn't look so good. What if you could create a smoother path, like the one below?

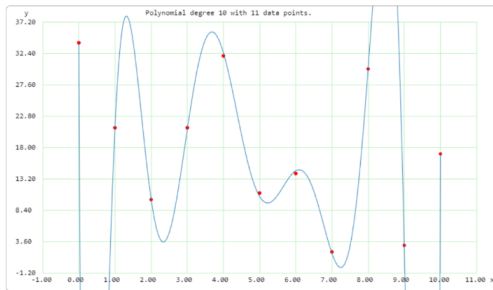


Representing Smooth Functions

What if I have hundreds of points?

If you have N points, you need a polynomial of degree $N - 1$ if you want the line to pass through all those points. But as you can see, when we have lots of points, the polynomial starts getting crazy on the extremes. We wouldn't want the character in our 2D game to go out of the screen while we're animating it, right?

Thankfully, someone took the time to solve this problem, because we have Bézier curves!



Source: <https://arachnoid.com/polysolve/>

Representing Smooth Functions

Bézier curves

A Bézier curves is a parametric curve (which means that all the coordinates of the curve depend on an independent variable t , between 0 and 1).

For example, given two points, we can calculate the linear B curve as the following interpolation:

$$B(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + tP_1$$



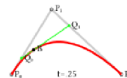
Given three points, we can calculate the quadratic Bézier curve that interpolates them.

$$Q_0(t) = (1 - t)P_0 + tP_1$$

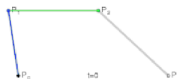
$$Q_1(t) = (1 - t)P_1 + tP_2$$

$$\begin{aligned} B(t) &= (1 - t)Q_0 + tQ_1 \\ &= (1 - t)[(1 - t)P_0 + tP_1] + t[(1 - t)P_1 + tP_2] \\ &= (1 - t)^2P_0 + 2(1 - t)tP_1 + t^2P_2 \end{aligned}$$

Source: [Wikipedia](https://en.wikipedia.org/wiki/B%C3%A9zier_curve)



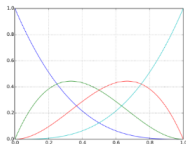
With four points, we can proceed with a similar reasoning.



Representing Smooth Functions

Bézier curves: going deeper

Yes, we can go deeper! If we have $n + 1$ points, we can find the n degree Bézier curve using the following formula



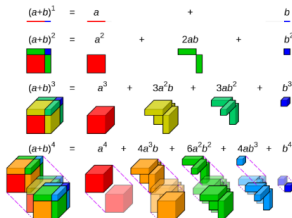
Blue: $b_{0,3}(t)$
Green: $b_{1,3}(t)$
Red: $b_{2,3}(t)$
Cyan: $b_{3,3}(t)$

Binomial coefficients

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

$$B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i = \sum_{i=0}^n b_{i,n}(t) P_i$$

Bernstein basis polynomials



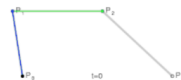
Source: [Wikipedia](#)

Representing Smooth Functions

From Bézier curves to B-Splines

If you have lots of points (say n), you need a Bézier curve with a degree $n-1$ to approximate it well, but that can be quite complicated computationally to calculate.

Someone wise thought: why don't we stitch together many Bézier curves between all these points, instead of one big Bézier curve that interpolates all of them?



Source: [Wikipedia](https://en.wikipedia.org/wiki/B%C3%A9zier_curve)

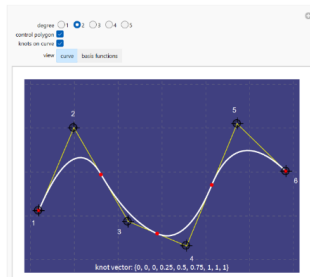
Representing Smooth Functions

B-splines in detail

A k -degree B-Spline curve that is defined by n control points, will consist of $n - k$ Bézier curves. For example, if we want to use a quadratic Bézier curve and we have 6 points, we need $6 - 2 = 4$ Bézier curves.

In this case we have $n=6$ and $k=2$

B-Spline Curve with Knots

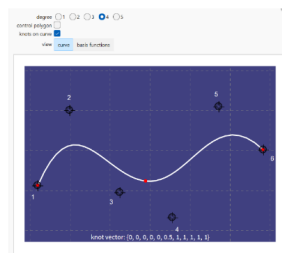
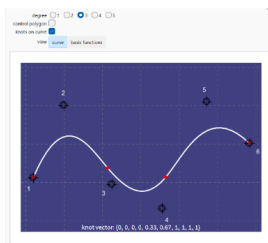
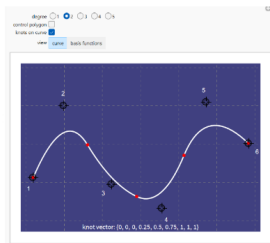


Source: [Wolfram Alpha](https://www.wolframalpha.com/)

Representing Smooth Functions

B-splines in detail

The degree of our B-Spline also tells what kind of continuity we get.



1.4.1 B-splines

An order k B-spline is formed by joining several pieces of polynomials of degree $k - 1$ with at most C^{k-2} continuity at the breakpoints.

Source: [MIT](#)

Representing Smooth Functions

Calculating B-splines: algorithm

A B-spline curve is defined as a linear combination of control points \mathbf{p}_i and B-spline basis functions $N_{i,k}(t)$ given by

$$\mathbf{r}(t) = \sum_{i=0}^n \mathbf{p}_i N_{i,k}(t), \quad n \geq k-1, \quad t \in [t_{k-1}, t_{n+1}] \quad (1.62)$$

In this context the control points are called *de Boor points*. The basis function $N_{i,k}(t)$ is defined on a *knot vector*

$$\mathbf{T} = (t_0, t_1, \dots, t_{k-1}, t_k, t_{k+1}, \dots, t_{n-1}, t_n, t_{n+1}, \dots, t_{n+k}) \quad (1.63)$$

where there are $n+k+1$ elements, i.e. the number of control points $n+1$ plus the order of the curve k . Each knot span $t_i \leq t \leq t_{i+1}$ is mapped onto a polynomial curve between two successive joints $\mathbf{r}(t_i)$ and $\mathbf{r}(t_{i+1})$. Normalization of the knot vector, so it covers the interval $[0,1]$, is helpful in improving numerical accuracy in floating point arithmetic computation due to the higher density of floating point numbers in this interval [133,300].

Given a knot vector \mathbf{T} , the associated B-spline basis functions, $N_{i,k}(t)$, are defined as:

$$N_{i,1}(t) = \begin{cases} 1 & \text{for } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \quad (1.58)$$

for $k=1$, and

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) \quad (1.59)$$

for $k > 1$ and $i = 0, 1, \dots, n$. These equations have the following properties [175].

Source: [MIT](#)

Representing Smooth Functions

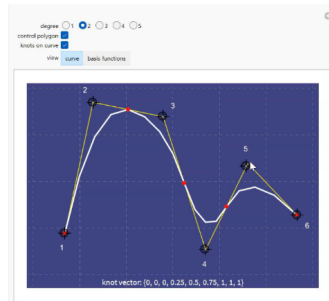
B-Splines: basis functions



Representing Smooth Functions

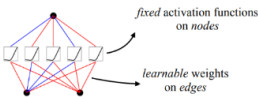
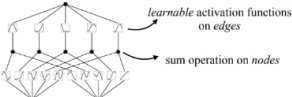
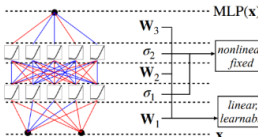
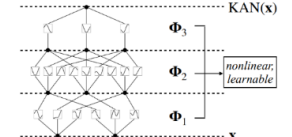
B-splines: local control

Moving a control point only changes the curve locally (in the proximity of the control point), leaving the adjacent Bezier curves unchanged!



- 1 Introduction to Data Fitting
- 2 Introduction to KAN**
- 3 KAT
- 4 Results

MLP vs KAN

Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(c)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) 	(b) 
Formula (Deep)	$\text{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\text{KAN}(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	(c) 	(d) 

Why KAN and not MLPs?

- **MLPs & Nonlinear Approximation:** MLPs are widely used for nonlinear function approximation due to their expressive power (Universal Approximation Theorem), but they have drawbacks: low interpretability, less accuracy in low dimensions, and the curse of dimensionality (COD) in high dimensions.
- **Introduction of KANs:** The paper introduces Kolmogorov-Arnold Networks (KANs), inspired by the Kolmogorov-Arnold representation theorem, as an alternative to MLPs.
- **Previous Work:** Earlier research focused on depth-2, width- $(2n + 1)$ networks, showing potential in breaking the COD empirically and theoretically, particularly with compositional function structures.
- **New Approach:** This paper proposes deeper and wider network architectures, leveraging backpropagation for training, to enhance performance beyond the traditional depth-2, width- $(2n + 1)$ representation.

KANs are a combo of MLPs and splines

- **Splines:** Accurate for low dimensions, easy to adjust locally and able to switch between different resolutions.(resolutions are the number of knots in the spline) But suffer from COD in high dimensions, unable to learn compositional structures
- **MLPs:** suffer less from COD thanks to their feature learning, but are less accurate than splines in low dimensions, because of their inability to optimize univariate functions.

To learn a function accurately, a model should not only learn the compositional structure (external degrees of freedom), but should also approximate well the univariate functions (internal degrees of freedom).

Solution: KANs = MLPs(learns external dofs) + splines(learns internal dofs)

Brief Idea of Advantages of KANs

- KANs can lead to accuracy and interpretability improvement over MLPs
- KANs can be made more accurate with grid extension
- KANs can beat the curse of dimensionality when there is a compositional structure in data, achieving better scaling laws than MLPs.
- PDE solutions can be approximated with KANs (Something we might want to research on)
- Two examples from mathematics (knot theory) and physics (Anderson localization) to demonstrate that KANs can be helpful “collaborators” for scientists to (re)discover math and physical laws. **No Background hence skipped**

- 1 Introduction to Data Fitting
- 2 Introduction to KAN
- 3 KAT
- 4 Results

Kolmogorov-Arnold Representation Theorem

Kolmogorov-Arnold representation theorem

Kolmogorov-Arnold representation theorem states that if f is a multivariate continuous function on a bounded domain, then it can be written as a finite composition of continuous functions of a single variable and the binary operation of addition. More specifically, for a smooth $f : [0, 1]^n \rightarrow \mathbb{R}$,

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

where $\phi_{q,p} : [0, 1] \rightarrow \mathbb{R}$ and $\Phi_q : \mathbb{R} \rightarrow \mathbb{R}$. In a sense, they showed that the only true multivariate function is addition, since every other function can be written using univariate functions and sum.

KAT

$$f(\mathbf{x}) = f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \phi_{q,p}(x_p) \right) \quad (2.1)$$

Handwritten notes: Red arrows point from the summation index q to the Φ_q term and from the summation index p to the $\phi_{q,p}$ term. Blue text next to the equation says: "two layer non lin." and "small no. of terms".

where $\phi_{q,p} : [0, 1] \rightarrow \mathbb{R}$ and $\Phi_q : \mathbb{R} \rightarrow \mathbb{R}$. In a sense, they showed that the only true multivariate function is addition, since every other function can be written using univariate functions and sum. One might naively consider this great news for machine learning: learning a high-dimensional function boils down to learning a polynomial number of 1D functions. However, these 1D functions can be non-smooth and even fractal, so they may not be learnable in practice [19][20]. Because of this pathological behavior, the Kolmogorov-Arnold representation theorem was basically sentenced to death in machine learning, regarded as theoretically sound but practically useless [19][20].

However, we are more optimistic about the usefulness of the Kolmogorov-Arnold theorem for machine learning. First of all, we need not stick to the original Eq. (2.1) which has only two-layer non-linearities and a small number of terms $(2n + 1)$ in the hidden layer: we will generalize the network to arbitrary widths and depths. Secondly, most functions in science and daily life are often smooth and have sparse compositional structures, potentially facilitating smooth Kolmogorov-Arnold representations. The philosophy here is close to the mindset of physicists, who often care more about typical cases rather than worst cases. After all, our physical world and machine learning tasks must have structures to make physics and machine learning useful or generalizable at all [21].

Mathematical Formulation

Let us introduce some notation. This paragraph will be a bit technical, but readers can refer to Figure 2.2 (left) for a concrete example and intuitive understanding. The shape of a KAN is represented by an integer array

$$[n_0, n_1, \dots, n_L], \quad (2.3)$$

where n_i is the number of nodes in the i^{th} layer of the computational graph. We denote the i^{th} neuron in the l^{th} layer by (l, i) , and the activation value of the (l, i) -neuron by $x_{l,i}$. Between layer l and layer $l + 1$, there are $n_l n_{l+1}$ activation functions: the activation function that connects (l, i) and $(l + 1, j)$ is denoted by

$$\phi_{l,j,i}, \quad l = 0, \dots, L - 1, \quad i = 1, \dots, n_l, \quad j = 1, \dots, n_{l+1}. \quad (2.4)$$

The pre-activation of $\phi_{l,j,i}$ is simply $x_{l,i}$; the post-activation of $\phi_{l,j,i}$ is denoted by $\tilde{x}_{l,j,i} \equiv \phi_{l,j,i}(x_{l,i})$. The activation value of the $(l + 1, j)$ neuron is simply the sum of all incoming post-activations:

$$x_{l+1,j} = \sum_{i=1}^{n_l} \tilde{x}_{l,j,i} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}), \quad j = 1, \dots, n_{l+1}. \quad (2.5)$$

Mathematical Formulation

$$\mathbf{x}_{l+1} = \underbrace{\begin{pmatrix} \phi_{l,1,1}(\cdot) & \phi_{l,1,2}(\cdot) & \cdots & \phi_{l,1,n_l}(\cdot) \\ \phi_{l,2,1}(\cdot) & \phi_{l,2,2}(\cdot) & \cdots & \phi_{l,2,n_l}(\cdot) \\ \vdots & \vdots & & \vdots \\ \phi_{l,n_{l+1},1}(\cdot) & \phi_{l,n_{l+1},2}(\cdot) & \cdots & \phi_{l,n_{l+1},n_l}(\cdot) \end{pmatrix}}_{\Phi_l} \mathbf{x}_l,$$

The breakthrough occurs when we notice the analogy between MLPs and KANs. In MLPs, once we define a layer (which is composed of a linear transformation and nonlinearities), we can stack more layers to make the network deeper. To build deep KANs, we should first answer: “what is a KAN layer?” It turns out that a KAN layer with n_{in} -dimensional inputs and n_{out} -dimensional outputs can be defined as a matrix of 1D functions

$$\Phi = \{\phi_{q,p}\}, \quad p = 1, 2, \dots, n_{\text{in}}, \quad q = 1, 2, \dots, n_{\text{out}}, \quad (2.2)$$

where the functions $\phi_{q,p}$ have trainable parameters, as detailed below. In the Kolmogorov-Arnold theorem, the inner functions form a KAN layer with $n_{\text{in}} = n$ and $n_{\text{out}} = 2n + 1$, and the outer functions form a KAN layer with $n_{\text{in}} = 2n + 1$ and $n_{\text{out}} = 1$. So the Kolmogorov-Arnold representations in Eq. (2.1) are simply compositions of two KAN layers. Now it becomes clear what it means to have deeper Kolmogorov-Arnold representations: simply stack more KAN layers!

Layer 2

5 input features, 1 output features
total of 5 functions to “learn”

Layer 1

2 input features, 5 output features
total of 10 functions to “learn”

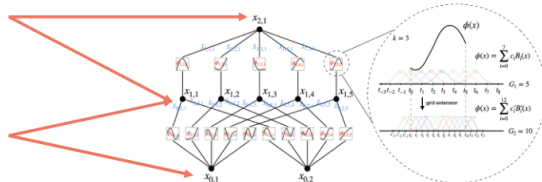


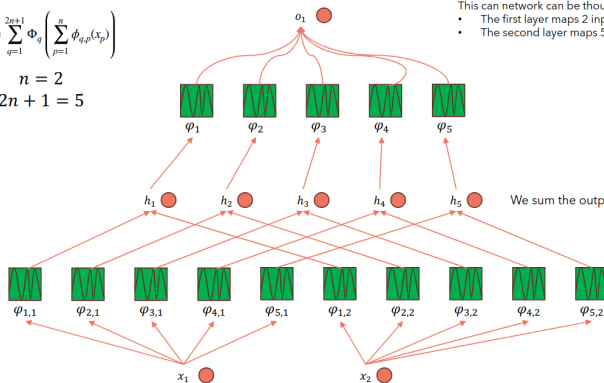
Figure 2.2: Left: Notations of activations that flow through the network. Right: an activation function is parameterized as a B-spline, which allows switching between coarse-grained and fine-grained grids.

Example KAN

Kolmogorov-Arnold Networks

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

$$n = 2$$
$$2n + 1 = 5$$



This can network can be thought of as two layers applied in sequence:

- The first layer maps 2 input features into 5 output features.
- The second layer maps 5 input features into 1 output feature.

We sum the output of the learnable functions

Instead of having learnable weights,
we have **learnable functions**

Example KAN

We can also rewrite the above equation to make it more analogous to Eq. (2.1), assuming output dimension $n_L = 1$, and define $f(\mathbf{x}) \equiv \text{KAN}(\mathbf{x})$:

$$f(\mathbf{x}) = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1, i_L, i_{L-1}} \left(\sum_{i_{L-2}=1}^{n_{L-2}} \cdots \left(\sum_{i_2=1}^{n_2} \phi_{2, i_3, i_2} \left(\sum_{i_1=1}^{n_1} \phi_{1, i_2, i_1} \left(\sum_{i_0=1}^{n_0} \phi_{0, i_1, i_0}(x_{i_0}) \right) \right) \right) \cdots \right), \quad (2.8)$$

which is quite cumbersome. In contrast, our abstraction of KAN layers and their visualizations are cleaner and intuitive. The original Kolmogorov-Arnold representation Eq. (2.1) corresponds to a 2-Layer KAN with shape $[n, 2n + 1, 1]$. Notice that all the operations are differentiable, so we can train KANs with back propagation. For comparison, an MLP can be written as interleaving of affine transformations \mathbf{W} and non-linearities σ :

$$\text{MLP}(\mathbf{x}) = (\mathbf{W}_{L-1} \circ \sigma \circ \mathbf{W}_{L-2} \circ \sigma \circ \cdots \circ \mathbf{W}_1 \circ \sigma \circ \mathbf{W}_0)\mathbf{x}. \quad (2.9)$$

It is clear that MLPs treat linear transformations and nonlinearities separately as \mathbf{W} and σ , while KANs treat them all together in Φ . In Figure 0.1 (c) and (d), we visualize a three-layer MLP and a three-layer KAN to clarify their differences.

↪ $Ax + b$

Implementation

Implementation details. Although a KAN layer Eq. (2.5) looks extremely simple, it is non-trivial to make it well optimizable. The key tricks are:

- (1) Residual activation functions. We include a basis function $b(x)$ (similar to residual connections) such that the activation function $\phi(x)$ is the sum of the basis function $b(x)$ and the spline function:

$$\phi(x) = w(b(x) + \text{spline}(x)). \quad (2.10)$$

We set

$$b(x) = \text{silu}(x) = x/(1 + e^{-x}) \quad (2.11)$$

in most cases. $\text{spline}(x)$ is parametrized as a linear combination of B-splines such that

$$\text{spline}(x) = \sum_i c_i B_i(x) \quad (2.12)$$

where c_i s are trainable. In principle w is redundant since it can be absorbed into $b(x)$ and $\text{spline}(x)$. However, we still include this w factor to better control the overall magnitude of the activation function.

- (2) Initialization scales. Each activation function is initialized to have $\text{spline}(x) \approx 0$ ^[2]. w is initialized according to the Xavier initialization, which has been used to initialize linear layers in MLPs.
- (3) Update of spline grids. We update each grid on the fly according to its input activations, to address the issue that splines are defined on bounded regions but activation values can evolve out of the fixed region during training^[3].

Parameter Count

Then there are in total $O(N^2L(G+k)) \sim O(N^2LG)$ parameters. In contrast, an MLP with depth L and width N only needs $O(N^2L)$ parameters, which appears to be more efficient than KAN. Fortunately, KANs usually require much smaller N than MLPs, which not only saves parameters, but also achieves better generalization (see e.g., Figure [3.1](#) and [3.3](#)) and facilitates interpretability. We remark that for 1D problems, we can take $N = L = 1$ and the KAN network in our implementation is nothing but a spline approximation. For higher dimensions, we characterize the generalization behavior of KANs with a theorem below.

Compared to MLP, we also have $(G+k)$ parameters for each activation, because we need to learn where to put the control points for the B-Splines.

$$\mathbf{r}(t) = \sum_{i=0}^n \mathbf{p}_i N_{i,k}(t), \quad n \geq k-1, \quad t \in [t_{k-1}, t_{n+1}]. \quad (1.62)$$

Approximation Theory, KAT

Theorem 2.1 (Approximation theory, KAT). *Let $\mathbf{x} = (x_1, x_2, \dots, x_n)$. Suppose that a function $f(\mathbf{x})$ admits a representation*

$$f = (\Phi_{L-1} \circ \Phi_{L-2} \circ \dots \circ \Phi_1 \circ \Phi_0)\mathbf{x}, \quad (2.14)$$

as in Eq. (2.7), where each one of the $\Phi_{l,i,j}$ are $(k+1)$ -times continuously differentiable. Then there exists a constant C depending on f and its representation, such that we have the following approximation bound in terms of the grid size G : there exist k -th order B-spline functions $\Phi_{l,i,j}^G$ such that for any $0 \leq m \leq k$, we have the bound

$$\|f - (\Phi_{L-1}^G \circ \Phi_{L-2}^G \circ \dots \circ \Phi_1^G \circ \Phi_0^G)\mathbf{x}\|_{C^m} \leq CG^{-k-1+m}. \quad (2.15)$$

Here we adopt the notation of C^m -norm measuring the magnitude of derivatives up to order m :

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

- 1 Introduction to Data Fitting
- 2 Introduction to KAN
- 3 KAT
- 4 Results**

- different themes
- different themes
- different themes
- different themes

End

MAS