POLITECNICO

MILANO 1863
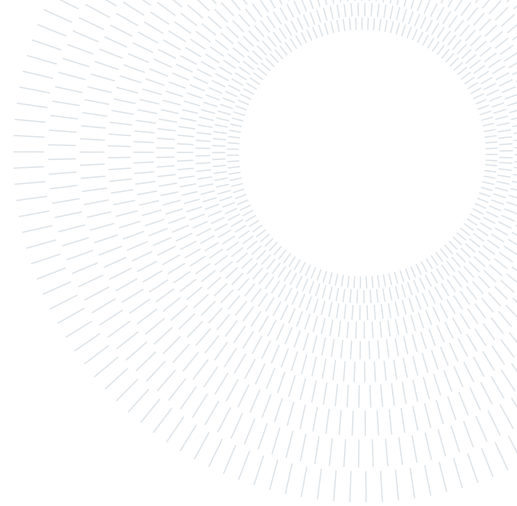
KANs: Kolmogorov-Arnold Networks

Pasqual Matteo Romilio, 10765765

Abstract: Inspired by the Kolmogorov-Arnold representation theorem, we propose Kolmogorov-Arnold Networks (KANs) as promising alternatives to Multi-Layer Perceptrons (MLPs). Unlike MLPs, which utilize fixed activation functions on nodes (*"neurons"*), KANs employ learnable spline-based activation functions on edges (*"weights"*), eliminating the need for linear weights. This design enables KANs to overcome MLPs in accuracy and interpretability. Empirically, smaller KANs achieve comparable or better accuracy than larger MLPs in function fitting, supported by faster neural scaling laws. Moreover, KANs excel in interpretability, allowing intuitive visualization and interaction, making them valuable collaborators for re-discovering mathematical and physical laws. By demonstrating superior performance in accuracy and usability, KANs offer a novel framework for enhancing modern deep-learning models that depend heavily on MLPs.

1. Introduction

Multi-layer perceptrons (MLPs), also known as fully-connected feedforward neural networks, are foundational building blocks of today's deep learning models. The importance of MLPs can never be overstated, since they are the default models in machine learning for approximating non-linear functions, due to their expressive power guaranteed by the universal approximation theorem.

However, are MLPs the best non-linear regressors we can build? Despite the prevalent use of MLPs, they have significant drawbacks. In transformers, for example, MLPs consume almost all non-embedding parameters and are typically less interpretable (relative to attention layers) without post-analysis tools. We propose a promising alternative to MLPs, called Kolmogorov-Arnold Networks (KANs).

Whereas MLPs are inspired by the universal approximation theorem, KANs are inspired by the Kolmogorov-Arnold representation theorem. Like MLPs, KANs have fully-connected structures. However, while MLPs place fixed activation functions on nodes "neurons"), KANs place learnable activation functions on edges ("weights"), as illustrated in Figure 1. As a result, KANs have no linear weight

matrices at all: instead, each weight parameter is replaced by a learnable 1D function parametrized as a spline. KANs' nodes simply sum incoming signals without applying any non-linearities.

One might worry that KANs are hopelessly expensive since each MLP's weight parameter becomes KAN's spline function. Fortunately, KANs usually allow much smaller computation graphs than MLPs embedding a 2-hidden-layer neural network.

Throughout this paper, we will show that KANs can lead to higher accuracy and improved interpretability over MLPs. We will show how, by leveraging the Kolmogorov-Arnold representation theorem, learnable activation functions and the absence of fixed-weight matrices allow KANs to efficiently capture complex relationships within the data[1].
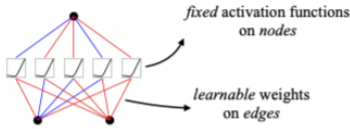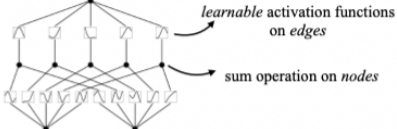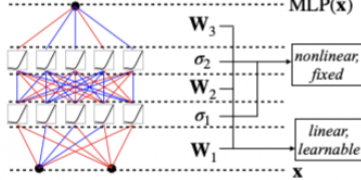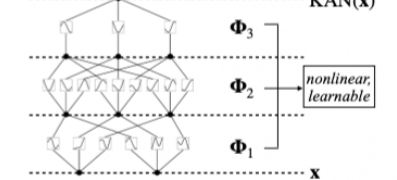
Model	**Multi-Layer Perceptron (MLP)**	**Kolmogorov-Arnold Network (KAN)**
Theorem	**Universal Approximation Theorem**	**Kolmogorov-Arnold Representation Theorem**
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) *fixed* activation functions on *nodes* / *learnable* weights on *edges*	(b) *learnable* activation functions on *edges* / sum operation on *nodes*
Formula (Deep)	$\mathbf{MLP(x)} = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\mathbf{KAN(x)} = (\mathbf{\Phi}_3 \circ \mathbf{\Phi}_2 \circ \mathbf{\Phi}_1)(\mathbf{x})$
Model (Deep)	(c) MLP(x) $\mathbf{W}_3$ σ_2 *nonlinear, fixed* $\mathbf{W}_2$ σ_1 *linear, learnable* $\mathbf{W}_1$ $\mathbf{x}$	(d) KAN(x) $\mathbf{\Phi}_3$ $\mathbf{\Phi}_2$ *nonlinear, learnable* $\mathbf{\Phi}_1$ $\mathbf{x}$

Figure 1: Multi-layer perceptrons (MLPs) vs Kolmogorov-Arnold Networks (KANs)

2. Kolmogorov-Arnold representation theorem

Whereas Multi-layer perceptrons (MLPs) are inspired by the universal approximation theorem, Kolmogorov-Arnold Networks (KANs) are inspired by the Kolmogorov-Arnold representation theorem [1].

Firstly, we start with the basic concept that a neural network can be described as a multivariate continuous function $f : \Re^n \to \Re^m$ that maps an input vector $\mathbf{x} \in \Re^n$ to an output vector $\mathbf{y} \in \Re^m$ via a series of computations. In particular, both MLPs and KANs can be described as a multivariate continuous function $f : \Re^n \to \Re$ that maps an input vector $\mathbf{x} \in \Re^n$ to an output value $y \in \Re$ via a series of computations: $f(x) \equiv KAN(x)$ [2, 3].

As in MLP when input variables are combined linearly we have to standardize inputs passing from a domain $D \subset \Re^n$ to a compact domain $D \subset [0, 1]^n$. A possibility is to apply an affine transformation to the data so that each feature will be normalized in $[0, 1] \in \Re$.

2.1. Theorem

We can now formally present and demonstrate the Kolmogorov-Arnold representation theorem.

The Kolmogorov-Arnold representation theorem is essential for proving that any neural network that has one output (for instance MLPs) can always be rebuilt as two-hidden-layer KANs. In particular, the 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 [4, 5].

Theorem 2.1 (Kolmogorov-Arnold representation theorem [4]). *Let f be an arbitrary multivariate continuous function on a bounded domain $f : [0,1]^n -> \Re$, then f:*

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n) = \sum_{q=1}^{2n+1} \Phi_q(\sum_{p=1}^{n} \phi_{q,p}(x_p))$$

with continuous one–dimensional inner functions $\phi_{q,p} : [0,1] -> \Re$ and continuous one–dimensional outer functions $\Phi_q : \Re -> \Re$.

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. The great advantage is that we have to learn only 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 with spline function, we can approximate their shapes and achieve the accuracy that we want [1].

2.2.   KAN matrix form

For a more formal definition, we define the representation matrix of the Kolmogorov-Arnold representation theorem ($f : [0,1]^n \to \Re$) [1]:

$$f(\mathbf{x}) = \boldsymbol{\Phi_{out}} \times \boldsymbol{\Phi_{in}} \times \mathbf{x}$$

where $\mathbf{x} \in \Re^n$, $\boldsymbol{\Phi_{in}} \in \Re^{2n+1,n}$, and $\boldsymbol{\Phi_{out}} \in \Re^{2n+1}$:

$$\boldsymbol{\Phi_{in}} = \begin{pmatrix} \phi_{1,1}(\cdot) & \dots & \phi_{1,n}(\cdot) \\ \vdots & & \vdots \\ \phi_{2n+1,1}(\cdot) & \dots & \phi_{2n+1,n}(\cdot) \end{pmatrix}, \quad \boldsymbol{\Phi_{in}} = (\Phi_1(\cdot) \dots \Phi_{2n+1}(\cdot))$$
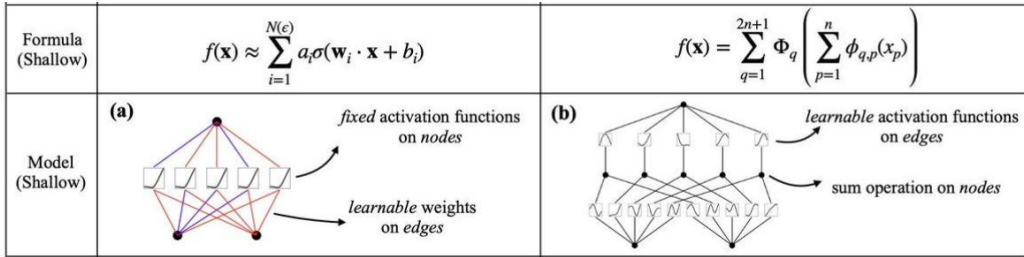

Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	**(a)** *fixed* activation functions on *nodes* / *learnable* weights on *edges*	**(b)** *learnable* activation functions on *edges* / sum operation on *nodes*

Figure 2: MLPs vs KANs: Shallow model

2.2.1   KAN layer form

We notice that both matrices $\boldsymbol{\Phi_{in}}$ and $\boldsymbol{\Phi_{out}}$ are special cases of the matrix $\boldsymbol{\Phi} \in \Re^{n_{out}, n_{in}}$ that represent a general Kolmogorov-Arnold layer.

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_{1,1}(\cdot) & \dots & \phi_{1,n_{in}}(\cdot) \\ \vdots & & \vdots \\ \phi_{n_{out},1}(\cdot) & \dots & \phi_{n_{out},n_{in}}(\cdot) \end{pmatrix}$$

so each layer map his input l $\mathbf{x}_l$ to the output $\mathbf{x}_{l+1}$ thought $\boldsymbol{\Phi}$:

$$\mathbf{x}_{l+1} = \boldsymbol{\Phi} \times \mathbf{x}_l \Rightarrow \mathbf{x}_{l+1} = \begin{pmatrix} \phi_{1,1}(\cdot) & \dots & \phi_{1,n_{in}}(\cdot) \\ \vdots & & \vdots \\ \phi_{n_{out},1}(\cdot) & \dots & \phi_{n_{out},n_{in}}(\cdot) \end{pmatrix} \mathbf{x}_l$$

Each layer has n_{in} inputs and n_{out} outputs and is mapped thought its respective layer matrix. In our case, the two layers are defined as follows:

- Layer 1: $\mathbf{\Phi_{in}}$ has $\mathbf{\Phi}$ with $n_{in} = n$ and $n_{out} = 2n + 1$
- Layer 2: $\mathbf{\Phi_{out}}$ has $\mathbf{\Phi}$ with $n_{in} = 2n + 1$ and $n_{out} = 1$

3. KAN

Kolmogorov-Arnold networks are more expressive than their basic formulation from the Kolmogorov-Arnold representation theorem. According to the theorem, the layer requirements are two layers, with a corresponding depth of $2n + 1$ and n. However, in practice, we can design networks with different topologies by adding layers or changing the definition of each layer [1]. The key idea is to stack layers from the input to the output.

3.1. Structure

With these premises, we can construct various topologies of Kolmogorov-Arnold networks by simply stacking layers and following these rules:
- A KAN, as all MPLs, is a directed acyclic graph (DAG).
- Each KAN layer is fully connected to the preceding and succeeding layers apart from the first layer will only have a succeeding layer and the last layer will only have a preceding layer.
- Each KAN layer is not connected with other layers
- Each KAN layers connection $\mathbf{x}_{l+1} = \mathbf{\Phi}_L \times \mathbf{x}_l$ is defined by $\mathbf{\Phi_L} \in \Re^{n_{out}, n_{in}}$ where n_{in} is the dimension of the $\mathbf{x}_l$ layer and n_{out} is the dimension of the $\mathbf{x}_l$ layer (Section 2.2).
- Every edge of al layer $\phi_{l,q,p}$ is associated with a one-dimensional activation function.

Let's say we have a KAN with L layers, where the lth layer has shape n_{l+1}, n_l and has formula $\mathbf{x}_{l+1} = \mathbf{\Phi} \times \mathbf{x}_l$. Then the whole network will be represented by:

$$KAN(\mathbf{x}) = \mathbf{\Phi_{L-1}} \times \cdots \times \mathbf{\Phi_1} \times \mathbf{\Phi_0} \times \mathbf{x}$$

We can also rewrite the above equation as a series of summations, assuming the output dimension $n_L = 1$ and the associated multivariate continuous function f: $f(x) \equiv KAN(x)$:

$$f(\mathbf{x}) = \sum_{i_{L-1}=1}^{n_{L-1}} \phi_{L-1, i_L, i_{L-1}} (...(\sum_{i_1=1}^{n_1} \phi_{1, i_2, i_1} (\sum_{i_0=1}^{n_0} \phi_{0, i_1, i_0}(x_{i_0}))...)$$

The network structure will be only a series of matrix-matrix multiplication of $\mathbf{\Phi_{L-1}}, \ldots, \mathbf{\Phi_1}, \mathbf{\Phi_0}$ while a MLPs was a series of linear transformations W_l and non-linear transformations σ:

$$MLP(\mathbf{x}) = \mathbf{W_{L-1}} \times \sigma \times \cdots \times \mathbf{W_1} \times \sigma \times \mathbf{W_0} \times \mathbf{x}$$
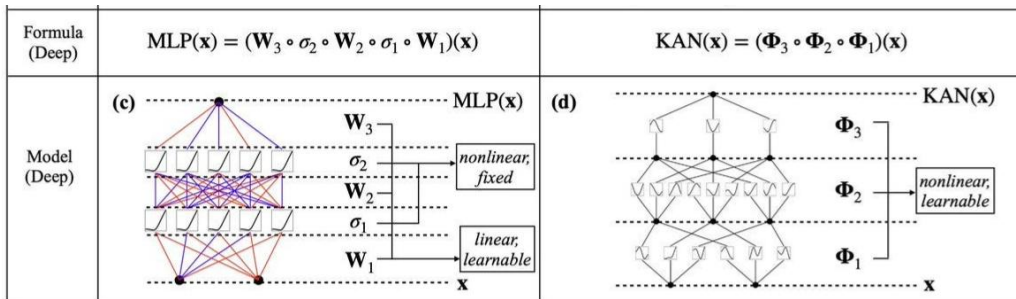

Figure 3: MLPs vs KANs: Deep model

Finally, a KAN can be conceptualized as a structured stack of KAN layers. Each KAN layer can be represented as a fully connected layer where every edge is associated with a one-dimensional (1D) function. The critical components that need to be defined are the representation and training of these activation functions which are the only learnable part of the function.

3.2. Activation functions

While the equation to compute, given an input, the KAN's output appears very simple, ensuring that the activation functions are well-trained is more complex. The most efficient technique involves leveraging spline functions. These functions are particularly effective for approximating the complexity and the non-linearities of the 1D functions associated with the KAN layers. This approach enhances both the flexibility and trainability of the network [1].

Formally speaking, in KANs, every activation function $\phi_{l,q,p}(\cdot)$ is defined as follows:

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

where:
- w_b and w_s are learnable weights; in principle are redundant since they can be absorbed into $b(x)$ and $spline(x)$, we still include these factors to better control the overall magnitude. In particular, w_s is very useful to scale the spline function.
- $b(x)$ is a fixed function called residual connection function defined by the silu function $b(x) = silu(x) = \frac{x}{1+e^{-x}}$. It is the counterpart of biases in MLPs.
- $spline(x)$ that is the learnable function where the real power of KANs came from. In most of the cases is parametrized as a linear combination of B-splines functions (B_i) and defining c_is the learnable B-splines points and k is the spline order $spline(x) = \sum_i^k c_i B_i(x)$.

3.3. Bézier and B-splines functions

Given the spline function now we have to define its shape. In literature, two main classes of functions are designed for KANs:
- Bézier functions which consider all the domains but are complex to train [6]
- B-splines functions which consider only a local domain but are easy to train [7].

The problem we are going to solve is that the spline function should pass through some tag points that will be adjusted during the training phase. To solve it let's consider a dual problem: imagine a character C must pass through n points $(P_1, \ldots, P_n)$. The most obvious way to traverse them is to go straight from P_i to P_{i+1} but this movement does not appear natural because we desire a smooth traversal movement that can be described as $(n-1)$-degree polynomial as in Figure 4.
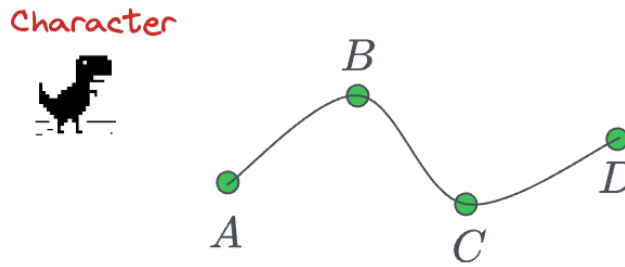

Figure 4: Example of 4-degree polynomial curve

The naive way to implement it is to define a function $h(x) : \Re \rightarrow \Re$:

$$h(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

then we can substitute points $(P_1, \ldots, P_n)$ into the function h and determine the values of the coefficients $(a_0, \ldots, a_{n-1})$. Now we realize that we have to solve N equations to determine the coefficients. Solving such a system of linear equations will be computationally expensive and almost infeasible for KANs. For this reason, we choose Bézier or B-splines functions.

3.3.1 Bézier

Bézier curves solve the problem more smartly. They provide a way to represent a smooth curve that passes near a set of control points without needing to solve a large system of equations as in Figure 5.
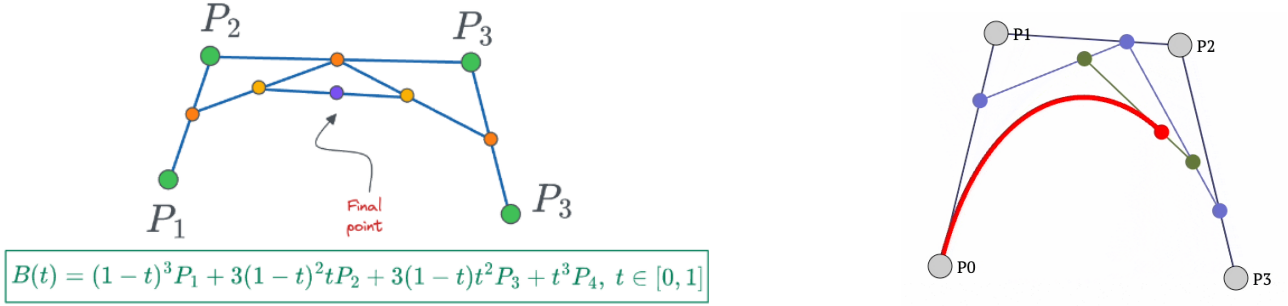

Figure 5: Example of 4-degree bezier curves

We define the Bézier curve $b(t) : \Re \to \Re$ noticing from low n polynomials as in Figure 5 that the coefficients match with the binomial coefficients of $(1+t)^n$. Then we can derive the binomial definition of the Bézier curve as:

$$\mathbf{b}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i, \quad t \in [0,1]$$

However, the problem is still the same as before. Having N data points will result in a polynomial of degree $N - 1$, which will be computationally expensive.

3.3.2 B-splines

B-splines provide a more efficient way to represent curves, especially when we deal with a large number of data. Unlike high-degree polynomials, B-splines use a series of lower-degree polynomial segments, which are connected smoothly. In other words, instead of extending Bézier curves to tens of hundreds of data points, which leads to an equally high degree of the polynomial, we use multiple lower-degree polynomials and connect them to form a smooth curve as in Figure 6.
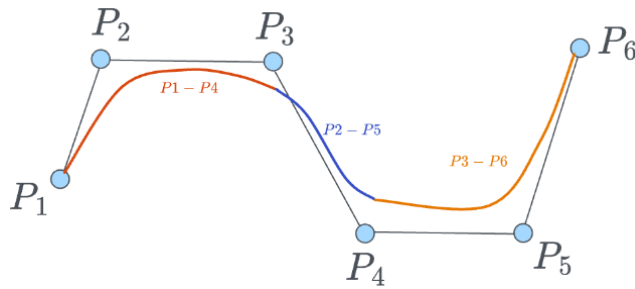

Figure 6: Example of 6-degree B-spline curve

When we have n control points and we create k degree polynomial Bézier curves, we get $(n-k)$ Bézier curves in the final Bsplines. We ensure also a certain continuity condition at the points where the curves meet:

- Position Continuity: C^0 Continuity
- Tangent Continuity: C^1 Continuity
- Curvature Continuity: C^2 Continuity

Similar to Bézier curves, we define the B-spline curve $B(x) : \Re \to \Re$ as a linear combination of the learnable position of points P_i and their associated non-learnable basis functions $N_{i,k}$:

$$\mathbf{B_i}(x) = \sum_{i=0}^{n} P_i N_{i,k} \Rightarrow \mathbf{splines}(x) = \sum_{i=0}^{k} c_i B_i(x) = \sum_{i=0}^{k} c_i \sum_{i=0}^{n} P_i N_{i,k}$$

Finally, we have defined spline as a B-spline function linear combination. This is best choice for KANs, even though they consider only a local domain, because their strength lies in their easy trainability, even as the number of points increases.

3.4. Initialization and training

After defining all the components of the network and their connections, we will explain how training is performed. The key idea behind the training process is to make the positions of the control points P_i in the activation function learnable, allowing the model to adapt and learn any arbitrary shape for the activation function that best fits the data. [1, 7]

3.4.1 Inizialization

The only trainable part of a KAN network is the set of activation functions. In particular, for each activation function, we will train the weights w_b and w_s and the spline function $spline(x)$.
- We initialize the scaling factor for the spline function at 1: $w_s = 1$.
- Each spline function is initialized with $spline(x) \approx 0$. This is done by drawing B-spline coefficients $c_i \sim \mathcal{N}(0, \sigma^2)$ with a small σ around 0.1. It's likely used to ensure symmetry and stability in early training stages.
- We initialize the scaling factor for the residual connection function with the Xavier initialization as in MLPs: $w_b \sim \mathcal{U}\left[-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right]$ where n_{in} and n_{out} are specific for any layers.

3.4.2 Training

Once the initialization has been defined, the KAN can be trained just like any other neural network. In particular for N_{epochs} we will perform:
- Forward propagation computing $KAN(\mathbf{x}) = \mathbf{\Phi_{L-1}} \times \cdots \times \mathbf{\Phi_1} \times \mathbf{\Phi_0} \times \mathbf{x}$
- Backword propagation computing the loss with the LSM or the cross-entropy method and then adjusting w_b and $splines(x)$ with some method as GD, SGD, NM, BFGS, or others. The parameter w_s, which is a scale factor, will only be modified if the spline activation values evolve out of the fixed region during training.

3.5. Hyperparameters and complexity

The hyperparameters of a Kolmogorov-Arnold network are:
- **L**: the depth of the KAN
- $\mathbf{N} \in \{\mathbf{n_0}, ... , \mathbf{n_l}\}$: width of each layer
- **k**: each spline is a linear combination of k B-splines. Usually, k is very small for instance $k = 3$.
- **G**: each B-splines has G control points

Then a KAN with hyperparameters **L,N,G,k**, considering N as the biggest $n_i \in N$, we will have in total $O(N^2 L(G + k)) \simeq O(N^2 LG)$ parameters. In contrast, an MLP with hyperparameters **L, N** needs only $O(N^2 L)$ 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 and facilitates interpretability [1].

3.5.1 Performance (Accuracy)

The most important Paper [1] has presented many performance-related results that compare the performance of KANs with MLP on various dummy/toy datasets. We will demonstrate that KANs are

more effective when we want to perform non-linear regression or PDE solving.

In Figure 7 the toy datasets are defined as follows:

1. $f(x) = J_0(20x)$ (Bessel function): Represented by a $KAN(L = 1, N = [1, 1])$.
2. $f(x, y) = e^{\sin(\pi x) + y^2}$: Represented by a $KAN(L = 3, N = [2, 1, 1])$.
3. $f(x, y) = xy$: Represented by a $KAN(L = 3, N = [2, 2, 1])$.
4. $f(x1, \ldots, x_{100}) = e^{\frac{1}{100} \sum_{i=1}^{100} \sin^2(\frac{\pi x_i}{2})}$: Represented by a $KAN(L = 3, N = [100, 1, 1])$.
5. $f(x_1, x_2, x_3, x_4) = e^{\sin(x_1^2 + x_2^2) + \sin(x_3^2 + x_4^2)}$: Represented by a $KAN(L = 3, N = [4, 4, 2, 1])$.

We train these KANs by increasing grid points every 200 steps, in total covering $G = \{3, 5, 10, 20, 50, 100, 200, 500, 1000\}$. We train MLPs with different depths and widths as baselines. Both MLPs and KANs are trained with LBFGS for 1800 steps in total.
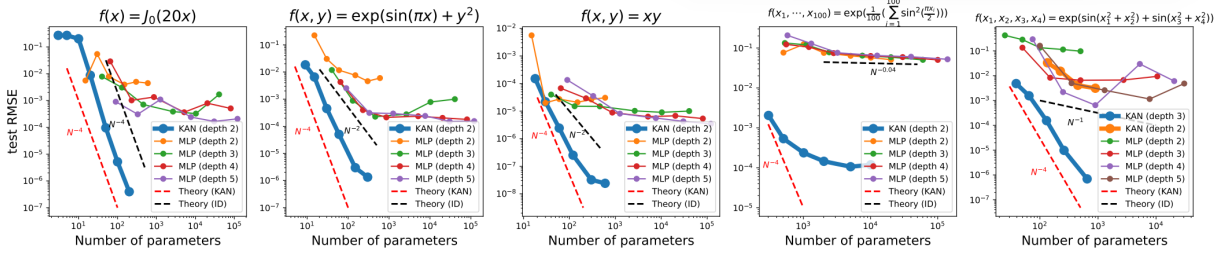

Figure 7: Results for KAN with toy dataset

In all the plots we can see that

- KANs consistently outperform MLPs, achieving significantly lower test loss across a range of parameters, and at much lower network depth (number of layers).
- KANs demonstrate superior efficiency, with steeper declines in loss, particularly noticeable with fewer parameters.
- MLP's performance almost stagnates with increasing the number of parameters.
- The theoretical lines N^{-4} for KAN and N^{-2} for ideal models (ID), show that KANs closely follow their expected theoretical performance.

3.5.2 Drawbacks

Currently, the biggest bottleneck of KANs lies in their slow training. KANs are typically 10 times slower than MLPs, given the same number of parameters (though KANs require fewer parameters than MLPs to accomplish the same task). This is primarily caused by the complex reshaping of spline(x), particularly on B-spline control points.

We should be honest that the first paper on KANs was pre-released in June 2024 [1], so at this point, no one is focused on optimization. This issue will likely be addressed as an engineering problem to be improved in the future rather than a fundamental limitation.
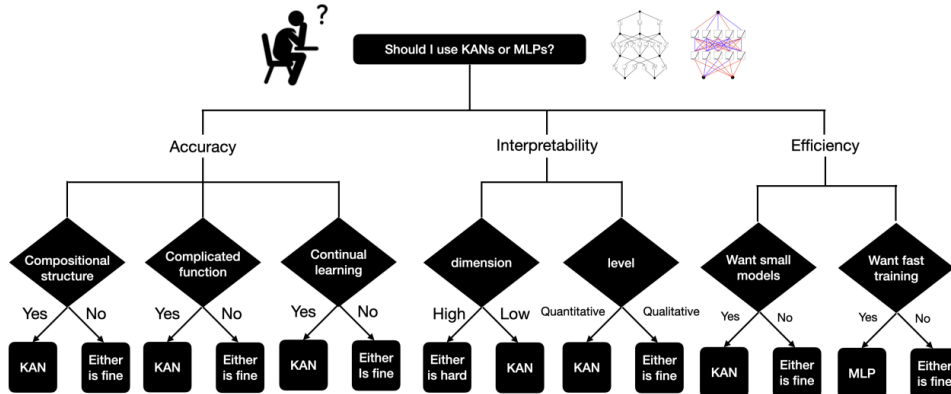

Figure 8: MLP vs KAN: decision tree

If one wants to train a model quickly, MLPs should be used. In other cases, however, KANs can be comparable to or even better than MLPs. The decision tree in Figure 8 can assist in determining when to use a KAN. In summary, if you value interpretability and/or accuracy and slow training is not a major concern, we suggest trying KANs.

3.6.   Advanced Techniques

Kolmogorov-Arnold networks can employ several advanced techniques to simplify the computation of the entire network reducing the time and space complexities. The main techniques are sparsification, pruning, and symbolification [1].

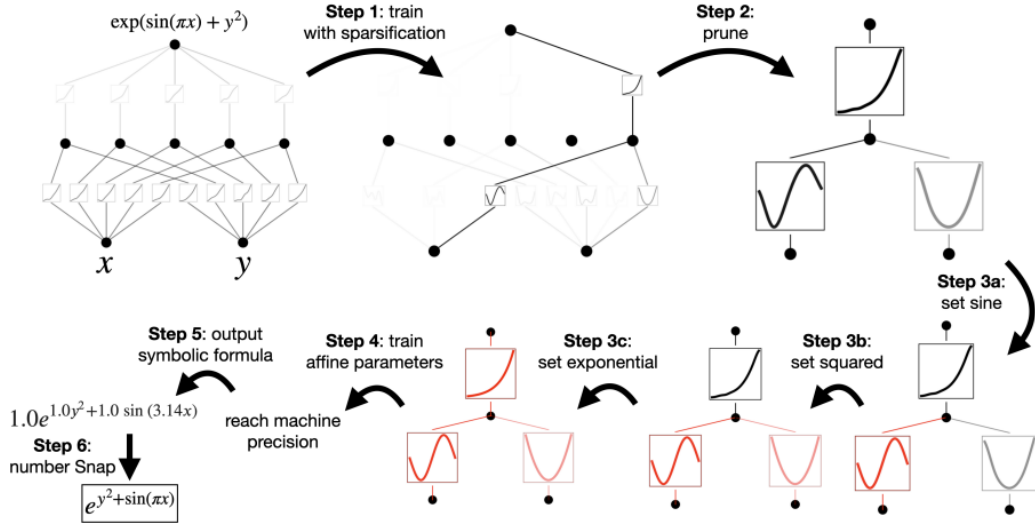Figure 9 presents an example of advanced techniques (sparsification, prunig, and Symbolification).

Figure 9: Advanced KAN techniques

3.6.1   Sparsification (Regularization)

For MLPs, regularization, particularly L1 regularization of linear weights, is used to promote sparsity, thereby improving predictions. KANs can adopt this high-level concept, but two modifications are necessary:

1. Since there are no linear weights in KANs, linear weights are replaced by learnable activation functions. Therefore, the L1 norm must be redefined for these activation functions.
2. Experimentally, we find that L1 regularization alone is insufficient for sparsifying KANs. Instead, additional entropy regularization encourages the model to use activation functions with lower complexity and fewer degrees of freedom.

We define the L1 norm of an activation function ϕ as its average magnitude over its N_p inputs: $|\phi|_1$.

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

Then for a KAN layer $\mathbf{\Phi}$ with n_{in} inputs and n_{out} outputs, we define the L1 norm of $\mathbf{\Phi}$ to be the sum of L1 norms of all activation functions: $|\mathbf{\Phi}|_1$.

$$|\mathbf{\Phi}|_1 \equiv \sum_{i=1}^{n_{in}} \sum_{j=1}^{n_{out}} |\phi_{i,j}|_1$$

9

In addition, as we have stated before, we define the regularization entropy of $\mathbf{\Phi}$ to be $S(\mathbf{\Phi})$.

$$S(\mathbf{\Phi}) \equiv -\sum_{i=1}^{n_{in}} \sum_{j=1}^{n_{out}} \frac{|\phi_{i,j}|_1}{|\mathbf{\Phi}|_1} \log(\frac{|\phi_{i,j}|_1}{|\mathbf{\Phi}|_1})$$

The total training objective l_{total} is the prediction loss l_{pred} plus L1 and entropy regularization of all KAN layers where we define μ_1, μ_2 are relative magnitudes usually set to $\mu_1 = \mu_2 = 1$, and λ controls overall regularization magnitude

$$l_{total} = l_{pred} + \lambda(\mu_1 \sum_{l=0}^{L-1} |\mathbf{\Phi_l}|_1 + \mu_2 \sum_{l=0}^{L-1} S(\mathbf{\Phi_l}))$$

3.6.2   Pruning

After training with the sparsification penalty, we may also want to prune the network to a smaller subnetwork. We sparsify KANs at the node level (rather than at the edge level). For each node (say the lth neuron in the lth layer), we define its incoming and outgoing scores as $I_{l,i}$ and $O_{l,i}$.

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

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

3.6.3   Symbolification

In some cases, after pruning, we suspect that some activation functions are very similar to symbolic functions (such as $\cos(x)$, $\tanh(x)$, e^x, $\ln(x)$, etc.). The goal of Symbolification is to provide an interface to set the symbolic ϕ to its specified symbolic form. We will see how this leads to interpretability in Section 3.7.

Practically, we include a function $fix\_symbolic(l, i, j, f)$ that can set the activation at position (l, i, j) to be f if they are similar. However, we cannot simply set the activation function to the exact symbolic formula, since its inputs and outputs may have shifts and scaling. Therefore, we obtain the pre-activation values x and post-activation values y from samples and fit affine parameters a, b, c, and d such that $y \approx cf(ax + b) + d$. The fitting is done by iterative search for a and b, followed by linear regression to find c and d.

3.7.   Interpretability

One of the biggest drawbacks of multilayer perceptrons is their lack of interpretability. On the other hand, Kolmogorov-Arnold networks are more interpretable since they learn univariate functions at all levels, which can be inspected if needed. It is relatively easy to determine the structure learned by the network using those formulas [7].

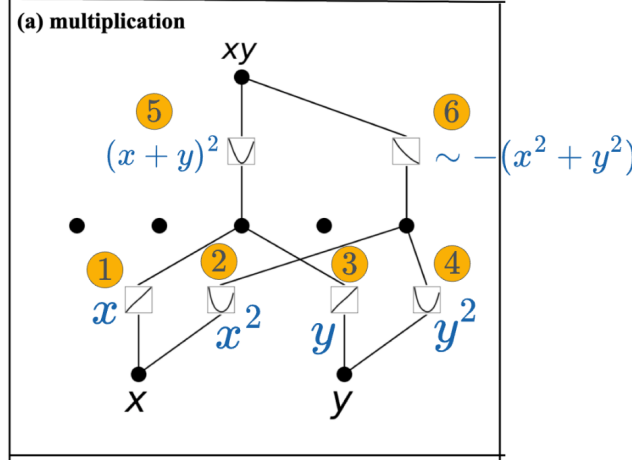Let's consider a KAN which is learning $f(x, y) = xy$ represented by $KAN(L = 2, N = [4, 2])$.

Figure 10: Interpretation of $f(x, y) = xy$

In Figure 10 inspecting the B-splines learned by KAN, we notice how the network is working.
In the first layer, both inputs x and y get transformed into their squares:

- Activation #1: map x to $x \Rightarrow \phi_{1,1}(k) = k$
- Activation #2: map x to $x^2 \Rightarrow \phi_{1,2}(k) = k^2$
- Activation #3: map y to $y \Rightarrow \phi_{1,3}(k) = k$
- Activation #4: map y to $y^2 \Rightarrow \phi_{1,4}(k) = k^2$

In the second layer, we take the previous output and combine it as follows:

- The sum of Activation #1 and #3, which is $(x + y)$, gets squared by activation #5 and have $(x + y)^2$ as output then $\phi_{2,1}(k) = k^2$.
- The sum of Activation #2 and #4, which is $(x^2 + y^2)$, gets negated by activation #6 and have $-(x + y)$ as output then $\phi_{2,2}(k) = -k$.

Finally, the last sum will return us $2xy$.

$$(x + y)^2 - (x^2 + y^2) = x^2 + y^2 + 2xy - x^2 - y^2 = 2xy$$

The interpretation is almost finished apart from the scaling factor. In the real networks the parameters w_s in the last layers are assigned to 0.5 then the last sum will return us xy.

$$\frac{1}{2}(x + y)^2 - \frac{1}{2}(x^2 + y^2) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + xy - \frac{1}{2}x^2 - \frac{1}{2}y^2 = xy$$

We have proved that the network can be very easy to interpret by extracting the meaning from $\phi_{i,j}(k)$. From the Paper [1] in Figure 11 we will see more examples of interpretability tasks.
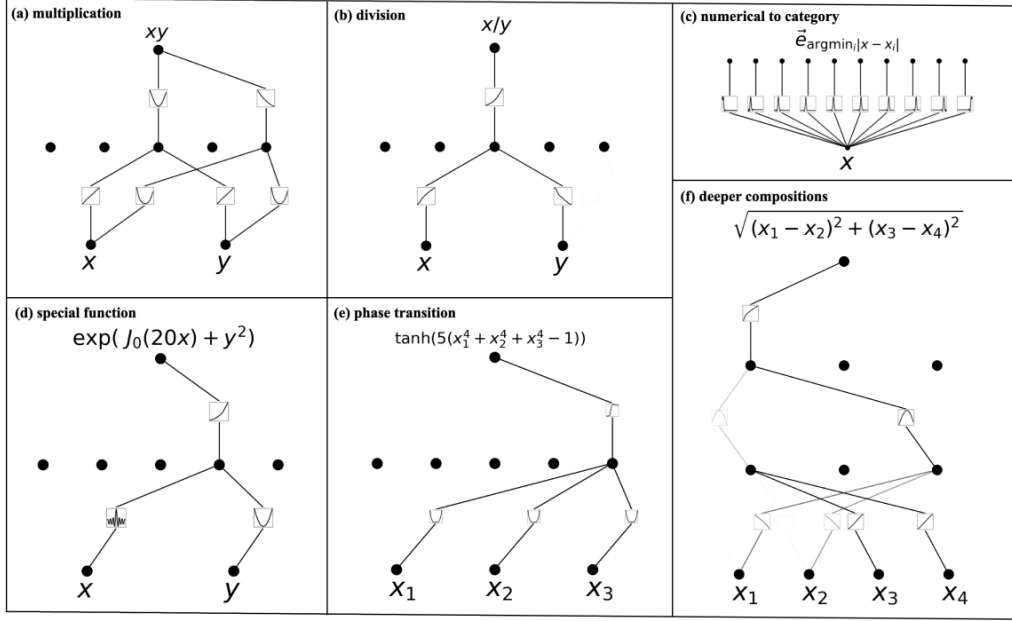
Figure 11: Interpretation of different functions

4. Code

5. Conclusions

Kolmogorov-Arnold Networks (KANs) offer an innovative approach to neural network architecture, diverging from Multi-layer Perceptrons (MLPs). Inspired by the Kolmogorov-Arnold representation theorem, KANs replace weight matrices and fixed activation functions with no weights and learnable spline-based functions that enhance expressiveness and adaptability. This unique design enables KANs to model complex, non-linear relationships with fewer parameters, leading to improved generalization and interpretability compared to MLPs. We have also demonstrated that KANs consistently outperform MLPs in non-linear regression and function approximation tasks, achieving lower test loss with reduced depth and complexity.

However, challenges persist in optimizing the training process for KANs. The reliance on spline activation functions introduces computational overhead, making KANs significantly slower than MLPs for comparable parameter counts. The current lack of mature optimization strategies for spline-based architectures further exacerbates this bottleneck. Despite these hurdles, advanced techniques such as sparsification, pruning, and symbolification have shown promise in reducing complexity, improving training efficiency, and providing a more interpretable network.

Looking forward, the potential for further improvement in KANs is immense. Research into accelerated training methods, more efficient spline representations, and scalable regularization techniques could unlock broader adoption and application of KANs across various machine learning tasks. Moreover, their ability to seamlessly integrate advanced techniques and provide symbolic interpretations of learned functions highlights their relevance in pushing the boundaries of interpretable AI.

KANs are not merely an alternative to MLPs but represent a paradigm shift toward more expressive, transparent, and efficient neural network architectures, with significant implications for the future of machine learning.

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.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] G. Strang. *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019.

[4] Jürgen Braun and Michael Griebel. On a constructive proof of kolmogorov's superposition theorem. *Constructive Approximation*, 30(3):653–675, Dec 2009.

[5] Johannes Schmidt-Hieber. The kolmogorov-arnold representation theorem revisited, 2021.

[6] M. R. Ndev. Understanding bézier curves, 2016. Accessed: 2024-11-25.

[7] Daily Dose of Data Science. A beginner-friendly introduction to kolmogorov–arnold networks (kan), 2023. Accessed: 2024-11-25.