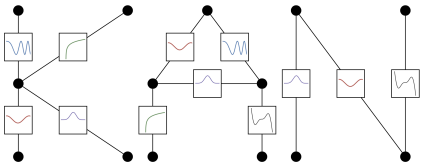
Theory-Inspired Architecture for Kolmogorov-Arnold Networks

Peter Hoffman[1]

[1]PhD student, LIDS, MIT



Welcome to the world of KANs

1 Overview

If the universal approximation serves as the theoretical foundation of Multi-Layer Perceptrons (MLPs), can other approximation theorems also offer inspiration for deep learning architectures? Building on a long string of results that ask similar questions, this blog offers additional evidence in the affirmative for the case of the Kolmogorov-Arnold (KA) representation theorem and the Kolmogorov-Arnold Networks (KANs) that it motivates. Our main technical insight is a theory-informed KAN architecture for the supervised approximation of functions motivated by physics and scientific-related tasks.

2 Representation theory background

The KA representation theorem states that any continuous function $f : [0, 1]^n \to \mathbb{R}$ admits the representation

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

where $\phi_{q,p} : [0, 1] \to \mathbb{R}$ and $\Phi_q : \mathbb{R} \to \mathbb{R}$ [1]. In other words, a multivariate function f can be written as a finite composition of univariate continuous functions.

The KA representation is not news to the machine learning community. Instead, results such as [5, 6, 7] have studied the KA representation to build neural networks as far back as 1993. Unfortunately, these analyses generally found the representation useless since the inner functions $\phi_{q,p}$ may be non-smooth and even fractal, making learning them infeasible in practice [8, 9]. However, these studies only examined the 2-layer $2n + 1$-width networks,

in accordance with the shape of the original theorem, and occurred before backpropagation and parallelization made computation more feasible.

One theoretical insight that gave hope to the ambition to use the KA representation for function learning came from David Sprecher [4], who proved that for any continuous function $f : [0,1]^n \to \mathbb{R}$, there exist real and monotonically increasing continuous functions $\phi_{q,p} : [0,1] \to \mathbb{R}$ with the Lipschitz property that

$$\ln\left(|\phi_{q,p}(x) - \phi_{q,p}(y)|\right) \leq \frac{\ln(2)}{\ln(2n+2)} \ln\left(|x-y|\right) \quad \forall x, y \in [0,1]. \tag{2}$$

In the original paper, these $\phi_{q,p}(x)$ functions are simply appropriate shifts and scalings of a specially constructed global inner function $\phi(x)$, which is then separately shown to have the desired Lipschitz property.

At an intuitive level, this result from Sprecher should come as good news since it guarantees the existence of KA representations where the inner functions are "nice" in the sense that they are non-fractal and do not have exploding gradients. However, despite offering a constructive proof of this fact, this result does not guarantee that a neural network will find this representation before getting stuck at a much worse (possibly fractal) one.

3 Kolmogorov-Arnold Networks

We now offer a brief overview of the mathematical formulation of deep KANs. While Multi-Layer Perceptrons (MLPs) learn linear weights between layers and use fixed activation functions at neurons, Kolmogorov-Arnold Networks (KANs) learn univariate activation functions and do not utilize weights. More formally, for a function $f(\mathbf{x}) : [0,1]^n \to \mathbb{R}$, an equivalent formulation of the KA theorem written in matrix form is

$$f(\mathbf{x}) = \Phi_{\text{outer}} \circ \Phi_{\text{inner}}(\mathbf{x}),$$

where for $\phi_{q,p} : [0,1] \to \mathbb{R}$ and $\Phi_q : \mathbb{R} \to \mathbb{R}$,

$$\Phi_{\text{inner}} = \begin{pmatrix} \phi_{1,1}(\cdot) & \cdots & \phi_{1,n}(\cdot) \\ \vdots & \ddots & \vdots \\ \phi_{2n+1,1}(\cdot) & \cdots & \phi_{2n+1,n}(\cdot) \end{pmatrix}, \quad \Phi_{\text{outer}} = \begin{pmatrix} \Phi_1(\cdot) & \cdots & \Phi_{2n+1}(\cdot) \end{pmatrix}.$$

We can then think of both Φ_{inner} and Φ_{outer} as individual instances of a more general function matrix Φ,

$$\Phi = \begin{pmatrix} \phi_{1,1}(\cdot) & \cdots & \phi_{1,n_{\text{inner}}}(\cdot) \\ \vdots & \ddots & \vdots \\ \phi_{n_{\text{outer}},1}(\cdot) & \cdots & \phi_{n_{\text{outer}},n_{\text{inner}}}(\cdot) \end{pmatrix}.$$

We can recover Φ_{inner} by setting $n_{\text{inner}} = n$, $n_{\text{outer}} = 2n+1$, and likewise we can recover Φ_{outer} by setting $n_{\text{inner}} = 2n+1$, $n_{\text{outer}} = 1$. In this notation, the original KA representation theorem in Equation 1 is simply a two-layer KAN.

If Φ defines a general functional layer, why not create deep networks simply by stacking Φ? Suppose we have L functional layers, then the whole network is

$$\text{KAN}(\mathbf{x}) = \Phi_L \circ \Phi_{L-1} \circ \cdots \circ \Phi_2 \circ \Phi_1(\mathbf{x}).$$

On an element-wise basis, let $x_{l,i}$ denote the pre-activation value of the i-th neuron in the l-th layer. If the width of layer l is n_l, then there are $n_l n_{l+1}$ learned activation functions

$\phi_{l,j,i}$ connecting neuron (l,i) to neuron $(l+1,j)$ in the fully connected mapping between layers l and $l+1$. Furthermore, the value of neuron $(l+1,j)$ is the sum of its incoming activation values:

$$x_{l+1,j} = \sum_{i=1}^{n_l} \phi_{l,j,i}(x_{l,i}) \quad \text{for } j = 1, \ldots, n_{l+1}.$$

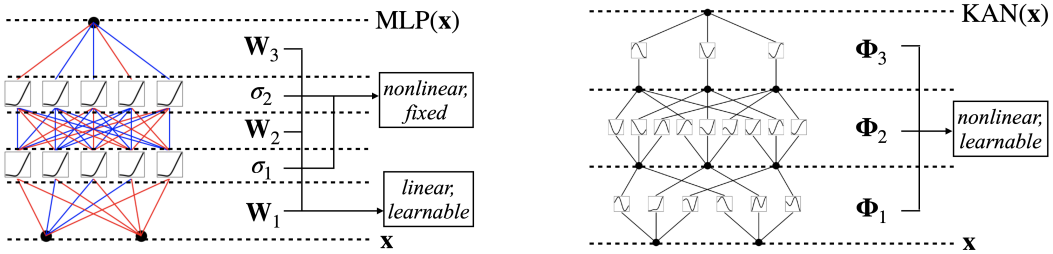
In contrast, a Multi-Layer Perceptron is interleaved by linear layers W_l and nonlinearities σ:

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

and takes the element-wise form

$$x_{l+1,j} = \text{ReLU}\left(\sum_{i=1}^{n_l} w_{l,j,i} x_{l,i}\right) \quad \text{for } j = 1, \ldots, n_{l+1}.$$

See Figure 2 for a pictorial comparison between MLP and KAN architectures.



(a) MLP architecture

(b) KAN architecture

Figure 2: MLPs use linear weights and fixed activation functions, while KANs use learned activation functions. Figure adapted from [2].

4 Implementation by Lui et al.

Recent work by Lui et al. developed the first trainable deep KAN in May 2024, attracting notable attention in the media and prompting a flurry of subsequent publications [2][10]. In this section, we first examine the practical elements of their implementation.

Activation function basis: In practice, we must make assumptions on the form of $\phi(x) : [0,1] \to \mathbb{R}$. In [2], Lui et al. use the parametrization

$$\phi(x) = w_a \, a(x) + b(x, \vec{w_b}),$$

where $a(x)$, $b(x, \vec{w_b}) : [0,1] \to \mathbb{R}$, the parameters w_a and $\vec{w_b}$ are trainable, and

$$a(x) = \frac{x}{1 + e^{-x}}.$$

Spline grid updates: The remaining component to be discussed is $b(x, \vec{w_b})$. In their implementation, Lui et al. parameterize $b(x, \vec{w_b})$ as a learnable linear combination of B-spline functions:

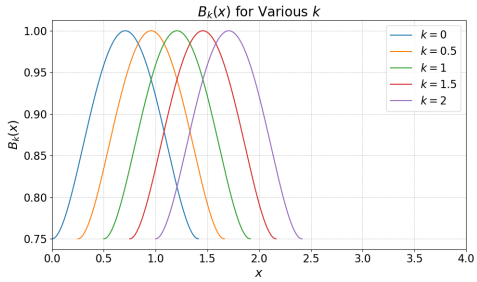
$$b(x, \vec{w_b}) = \sum_k (\vec{w_b})_k \, B_k(x),$$

where these B-splines B_k are stored in memory and essentially used as basis functions. An example of one such family of basis functions is

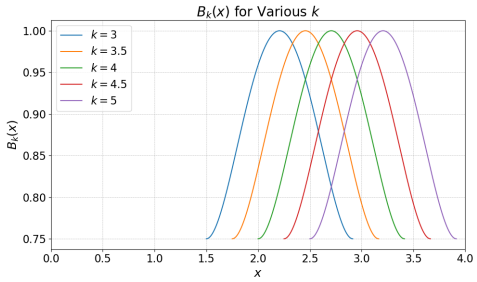
$$B_k(x) = \left(\left(x - \sqrt{0.5} + \frac{k}{2}\right)^4 - \left(x - \sqrt{0.5} + \frac{k}{2}\right)^2\right) + 1, \quad x \in \left[\frac{k}{2}, \sqrt{2} + \frac{k}{2}\right],$$

for all $k \in \mathbb{R}$. Figure 3a shows five functions from this family for $k = 0,\ 0.5,\ 1,\ 1.5,\ 2$.

However, this introduces the problem that spline functions are defined over closed intervals of x, but neuron values evolve unpredictably during training. To address this, [2] defines a closed grid over the input domain of pre-activation neuron values and updates these grid boundaries at each iteration during training. More concretely, although B_k is defined for all $k \in \mathbb{R}$, at each iteration we only utilize a subset of B_k. If at some iteration x exits the subset of $\mathbb{R}$ covered by the joint domain of these B_k, then we update the spline grid by shifting the indices k in our basis so that the joint domain of the new basis includes the appropriate range of pre-activation values of that neuron. We provide an example of shifting the basis below in Figure 3.



(a) Original B-spline basis using $k = 0,\ 0.5,\ 1,\ 1.5,\ 2$



(b) Updated B-spline basis using $k = 5,\ 6,\ 7,\ 8,\ 9$

Figure 3: Updating spline grids used in the basis of $\phi(x)$ to account for varying neuron values of x.

5 KANs ≈ MLPs?

On one hand, KANs are essentially MLPs with B-spline activation functions. Why is this the case? While KANs do not utilize traditional linear weights between layers—instead "learning" all non-linear functions $\phi_{l,j,i}$ that map between neuron i in layer l and neuron j in layer $l+1$—the linear weights w_a and $\vec{w_b}$ reside within these functions $\phi_{l,j,i}$:

$$\phi_{l,j,i} = w_a a(x) + \sum_k (\vec{w_b})_k B_k(x).$$

In this view, $\phi_{l,j,i}$ is simply a linear combination of non-linear functions of x: precisely a layer in an MLP. The implication of this is that for every fixed KAN (e.g., post-training), there exists an MLP that achieves the same mapping from input to output since each function in a KAN is itself an MLP using non-linear activation functions $a(x)$ and $B_k(x)$. In Figure 4 below, we explicitly demonstrate this correspondence for the case of a single $\phi(x)$ with two basis functions $B_1(x)$ and $B_2(x)$.

This is only part of the story, however, since the number of B_k changes during training as the grid intervals are updated and the number of possible B_k is infinite. Therefore, while it is

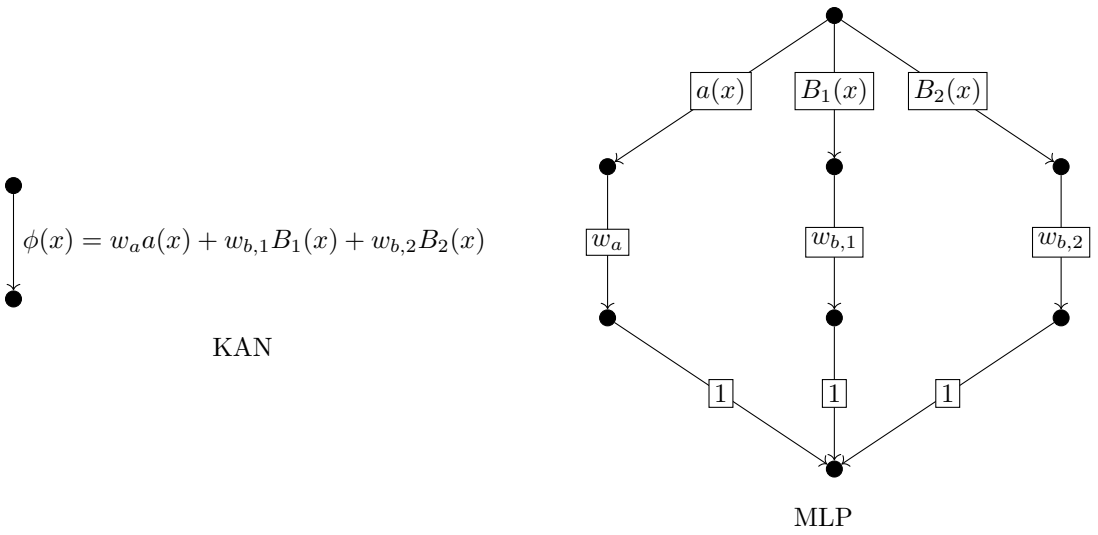


Figure 4: An equivalent formulation of a KAN as an MLP, where we first apply fixed nonlinear activation functions, then take learnable scaler weightings, then sum.

possible to take any fixed KAN and write down an equivalent MLP as previously described, *during training* KANs are flexible in ways MLPs are not since they update the current basis B_k at each iteration.

KANs' use of B-spline functions for the functional basis introduces obvious computational costs that MLPs with ReLU activation functions avoid. Yet there are also benefits. First, a linear combination of B-spline functions can be made more accurate to a target function by making the spline grid arbitrarily fine-grained. More formally, for $B_k(x)$ defined above, we can increase the spline grid granularity by instead using

$$B_{k,g}(x) = \left(\left(x - \sqrt{0.5} + \frac{k}{g}\right)^4 - \left(x - \sqrt{0.5} + \frac{k}{g}\right)^2\right) + 1, \quad x \in \left[\frac{k}{g}, \sqrt{2} + \frac{k}{g}\right],$$

where increasing the granularity $g \in \mathbb{R}$ decreases the offset of these functions along the x-axis and thus increases the granularity of the spline grid. In contrast, MLPs do not have any parameter that can be tuned to increase or decrease the granularity of the approximation during training.

6 Theory-informed KAN Architecture

As is the case with many representation results, Theorem 1 guarantees the *existence* of Φ_{inner} and Φ_{outer}, but says nothing about how easy it is to find them via optimization. In this section, we ask the natural question of whether choices related to KAN architecture affect the learnability of this representation.

The original paper, which used deep fully connected networks, has spurred significant research since its publication in May 2024. However, nearly all of this subsequent research has fallen along two primary axes. The first concerns optimization, with the goal of improving the admittedly slow training of the original KAN Python package Pykan. This research, including [14] and [15], aims to borrow many of the optimization methods from MLPs that

have resulted in fast, scalable, and parallelizable learning. The second primary area of recent research has been applying KANs in domain-specific settings, such as computer vision [11, 12, 13] and learning physics-related PDEs [16, 17].

Despite this stream of research on KAN optimization and applications, research on architectures beyond the deep, fully connected networks used in [2] is lacking. We aim to fill this gap by presenting evidence that sparse networks—inspired by what we refer to as "KAN recurrence"—achieve lower training and test loss when tasked with the supervised learning of physics-inspired functions.

We first ask if any of the aforementioned theoretical results implicitly suggest architectures that make the KA representation of a function easier to learn. Consider the result by Sprecher in Equation 2, which bounds the Lipschitz property of the constructed Φ_{inner} by $\frac{\ln(2)}{\ln(2n+2)}$. In an intuitive sense, smooth functions should be easier to learn because functional behavior between points in the training set is controlled by the gradient. In other words, each layer of our network, which is an approximation for KA representation, can more safely interpolate between data points in the training set. Therefore, Sprecher's result seems to suggest that increasing n, the input dimension and the parameter that controls the width of the network, should result in KA representations that are easier to learn via a KAN.

Secondly, what if we apply the KA theorem on itself? After all, the KA theorem decomposes an arbitrary function f into a sum of functions Φ_q and $\phi_{q,p}$ that are hopefully easier to learn. Yet we've already covered research that suggests that these learned functions can be pathological and thus difficult to find via optimization and gradient descent. As a solution, what if we re-apply the KA theorem on these $\phi_{q,p}(x_p)$ for $x_p \in [0,1]$? In a sense, this would decompose these learned functions into even simpler pieces. While an intuitively pleasing idea, the obvious objection is that $\phi_{q,p}(x_p)$ take one-dimensional inputs, yet the KA theorem only holds for functions of at least two variables. However, we can use the projection $x_p \mapsto (x_p, \ldots, x_p) \in \mathbb{R}^e$ to circumvent this issue, which simply repeats x_p a total of e times. In this new setting, we simply apply the usual KA theorem in Equation 1 on the e-variable function $\phi_{q,p}(x_p, \ldots, x_p)$:

$$\phi_{q,p}(x_p) = \phi'_{q,p}(x_p, \ldots, x_p) = \sum_{q'=1}^{2e+1} \Phi_{q'} \left(\sum_{p'=1}^{e} \phi_{q',p'}(x_{p'}) \right). \tag{3}$$

We can then plug Equation 3 into the main KA theorem in Equation 1 to obtain

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

which gives the full KA-KA decomposition of $f(\mathbf{x})$. A depiction of the corresponding architecture is shown in Figure 4.

We can then plug in Equation 3 into the main KA theorem in Equation 1 to obtain

$$f(\vec{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi'_{q,p}(x_p, x_p) \right) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \left(\sum_{q'=1}^{2(2)+1} \Phi_{q'} \left(\sum_{p'=1}^{2} \phi_{q',p'}(x_p) \right) \right) \right),$$

which gives the full KA-KA decomposition of $f(\vec{x})$. A depiction of the corresponding architecture is shown in Figure 5.

At this point in our analysis, we'd like to highlight the boundary between theory and practice. As stated, our underlying goal was to re-apply the KA theorem on these inner

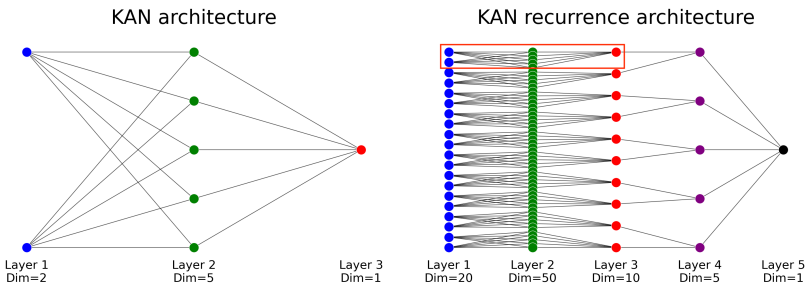


Figure 5: (LEFT) The original KAN architecture represents an $n = 2$ dimension function using $2n + 1 = 5$ hidden nodes. Therefore, there are $(n)(2n + 1) = 10$ learned $\phi(x)$ in the first layer and $1 \cdot (2n + 1) = 5$ learned Φ in the second layer. (RIGHT) In the KAN recurrence architecture, we represent each of these 10 learned $\phi(x)$ using a second KA decomposition with expansion $e = 2$, shown in the red box. Notice that each of these inner KA decompositions, one of which shown in the red box, has the same architecture as the original KAN architecture on the left since we set the expansion e equal to the input of the original function f.

functions $\phi(x)$. However, in doing so, we've clearly exited the preconditions required by the KA theorem. While we've projected $x \to (x, x)$, the functions $\phi'_{q,p}(x_p, x_p)$ aren't technically a function of two variables since it's domain is trivially a one dimensional subspace of $\mathbb{R}^2$.

In light of this input dimension-related technicality, one might wonder why we bothered projecting $x \to (x, x)$ in the first place. After all, if the space spanned by (x, x) is a one-dimensional subset of $\mathbb{R}^2$, what did we actually achieve by projecting in the first place? Referring again to Figure 5, we observe that the width of the first layer of the KAN recurrence architecture (RIGHT) is given by $(n)(2n + 1)e$ where $e = 2$ is the expansion of the input and n is the input dimension of the represented function f. For the curious reader, this came from the fact that we originally had $(n)(2n + 1)$ functions ϕ which we represented individually on input dimension of e. Similarly, the width of the second layer is $(2e + 1)(n)(2n + 1)$. Therefore, we can increase the width of these first and second layers by increase our expansion. Recalling our practical insights from Sprecher's bound on the Lipschitz property of the constructed ϕ [4], it becomes clear why setting $e > 1$ is useful: it increases the width of the network, thus potentially tightening the bound on the Lipschitz property, and thus hopefully resulting in a functional representation that is easier to learn via gradient descent.

To recap the main insights from this section, we decomposed the KA representation of a function by reapplying the KA theorem on the ϕ after an expansion of the input dimension. We then reasoned that, although this procedure potentially violates the preconditions used in the KA theorem, this expansion is still useful since it creates wider networks, which are desirable in light of the Lipschitz bound by [4].

7 Architecture-related Experiments

In this section, we test the above claims regarding the impact of width and KA recurrence on the supervised learning of functions motivated by physics and scientific-related tasks.

7.1 Dataset Generation

Suppose we wish to approximate the functions

$$f_1(x_1, x_2) = x_1 x_2$$
$$f_2(x_1, x_2) = \cos\big(\sin(x_1 + x_2) + x_1 + x_2\big)$$
$$f_3(x_1, x_2) = e^{\sin(x_1)+x_2}.$$

While these functions are admittedly low-dimensional and simplistic compared to other functions encountered in areas such as text and images, they will serve as a proof of concept for our proposed architecture. Additionally, they include components, such as trigonometric and exponential functions, which are often encountered in physical and engineering-related domains and for which shallow and narrow MLPs often have difficulty approximating. These target functions are shown below in Figure 6.

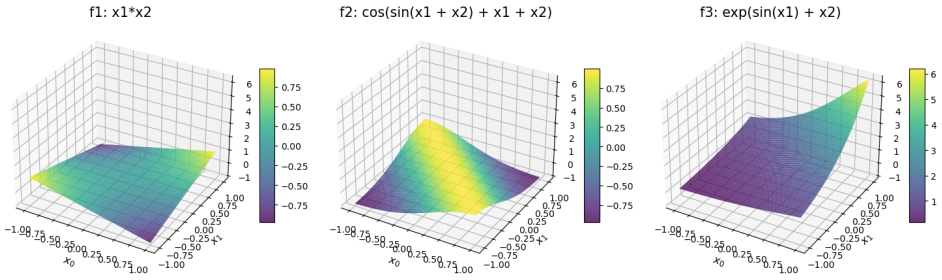


Figure 6: Visualization of target functions used for supervised approximation experiments.

For our training dataset generation, we drew 1000 uniformly random points from $[-1, 1]^2$ and proceeded to evaluate each of the functions on this 1000×2-dimensional vector, producing a vector of labels of size 1000×1 for each of our functions. For our validation dataset, we ran the same procedure.

7.2 Architecture

We now turn to our architecture for the experiment. The question we seek to answer is whether the KA-recurrence architecture derived above in Figure 5 offers any advantages over traditional fully connected architectures. To this end, we introduce the three networks we will examine. The first is the KAN recurrence architecture derived above with an expansion $e = 2$. The second architecture is a fully connected KAN with 6 hidden layers and 5 nodes per hidden layer. The third architecture is a fully connected KAN with 3 hidden layers and 10, 10, 4 nodes per hidden layer, respectively. It is trivial to check that each network has the same number of functions ± 1, and thus the same number of parameters[1].

7.3 Training

Since the deep KAN architecture is still very new, only having been first introduced in May 2024, the only existing Python package was Pykan. However, we found these methods generally unsuitable for our needs since they did not support masking or integration with MLPs

[1]Function totals across networks. Architecture 1: $100 + 50 + 10 + 5 = 165$. Architecture 2: $10 + 6(25) + 5 = 165$. Architecture 3: $20 + 100 + 40 + 4 = 164$.

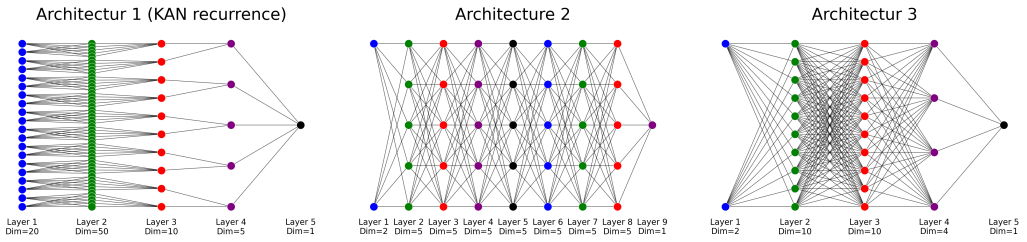


Figure 7: Architecture 1 is a KAN recurrence network with expansion $e = 2$. Architecture 2 is a 6-layer fully connected KAN with width 5. Architecture 3 is a 3-layer fully connected KAN with layer widths 10, 10, and 4, respectively.

with weight sharing, which was our initial idea for this project. Therefore, we coded the KAN net and layer classes ourselves while maintaining compatibility with backpropagation and used the Adam optimizer.

As opposed to polynomial basis functions, we used the trigonometric basis functions

$$B_k(x) = \cos^k(x) \tag{4}$$

$$B_{k+1}(x) = \sin^k(x) \tag{5}$$

for $k = 0, 2, 4, 6, \ldots$. We found experimentally that this produced more stable results than spline basis functions, avoided the necessity of grid interval updates, yet still allowed the option to increase basis granularity during training.

We trained each network locally on the GPU for 50 epochs, with a batch size of 10, a learning rate of 10^{-3}, the Adam optimizer, and used the Mean Squared Error (MSE) loss. The total training time across all functions and all architectures was less than 10 minutes.

7.4 Experimental results

We first discuss the training loss, shown in Figure 8. IN the initial epochs, the rate of training loss descent was consistent across Architecture 1 (BLUE), Architecture 2 (ORANGE), and Architecture 3 (GREEN) for every function investigated. However, after about epoch 15 for each function, Architectures 2 and 3 began to plateau while Architecture 1 continued to decrease. After 50 epochs, it was clear across all functions that Architecture 1 had achieved a lower training loss than Architectures 2 and 3. We also speculate, given the right-tail behavior of the blue lines, that the training loss of Architecture 1 would have continued to decrease if given additional epochs. Furthermore, it is encouraging that all three trend lines are smooth and non-intersecting, suggesting that the optimization process is free from unstable learning rates or exploding gradients.

Next, Table 1 below shows the final validation loss for each architecture-function pair. For each function $f_1,\ f_2,\ f_3$, Architecture 1 achieved the lowest validation loss by a wide margin. Meanwhile, Architectures 2 and 3 had similar validation losses but with Architecture 3 (wide, fully connected) slightly outperforming Architecture 2 (deep, fully connected) in every case.

We now present the plots containing Actual vs. Predicted function values for each of the nine architecture-function pairs in Figure 9. Given the training and validation data above, it is unsurprising that Architecture 1 demonstrated the best learning across all functions. More

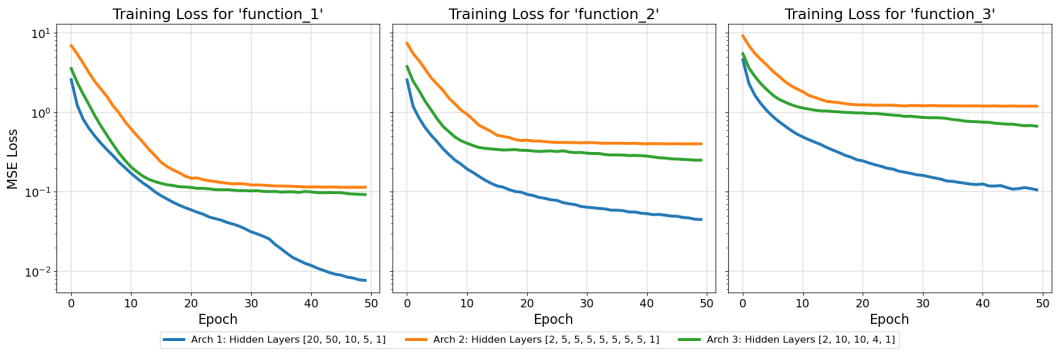


Figure 8: Training loss across epochs for each architecture and function. In every function, Architecture 1 (BLUE) achieves a lower training loss than Architecture 3 (GREEN), which in turn outperforms Architecture 2 (ORANGE).

	Arch1 (Nested KAN)	**Arch2** (FC Deep)	**Arch3** (FC Wide)
TF1 $f = x_1 x_2$	0.0077	0.1071	0.0933
TF2 $f = \cos(\sin(x_1 + x_2) + x_1 + x_2)$	0.0559	0.3784	0.2904
TF3 $f = e^{\sin(x_1)+x_2}$	0.1180	1.1303	0.7063

Table 1: Validation loss for each architecture-function pair. For each function considered, Architecture 1 achieves the lowest validation loss by a wide margin, followed by Architecture 3, then Architecture 2.

notable is that Architectures 2 and 3 almost didn't learn at all. Furthermore, additional training runs produced similar results.

Another interesting artifact of these Actual vs. Predicted plots is that the functional basis used in the KAN architecture has an unclear relationship with performance on certain functions. For example, we anticipated that Function 2, $f = \cos(\sin(x_1 + x_2) + x_1 + x_2)$, would have been easiest to learn since it is a direct composition of functions in the functional basis (recall we used a trigonometric basis defined in Equations 4 and 5). However, the lowest training and test loss was for Function 1, $f = x_1 x_2$. While we make no claims about the optimal basis among B-spline, polynomial, or trigonometric, we do claim that these results suggest that the answer is more complex than simply choosing the basis that resembles the function you think you're trying to learn.

7.5 Main Findings

When conducting small-scale machine learning, such as function learning of physics-inspired processes from which data has been collected, perhaps the most obvious architecture would be a fully connected network that balances depth and width. However, put lightly, the above results suggest that this may not be the best choice, and put more strongly, the above results suggest that fully connected architectures might cause you to learn nothing at all.

Recall that we motivated this architecture by invoking theory provided by [4] and a derivation of KA recurrence. To this end, it is pleasing that our experiments offer some evidence that theoretical intuition can indeed support success in practice.

Lastly, it's worth considering what features of Architecture 1 contributed to its success. First, we expect that sparsity plays a role. From separate outside research on KANs un-

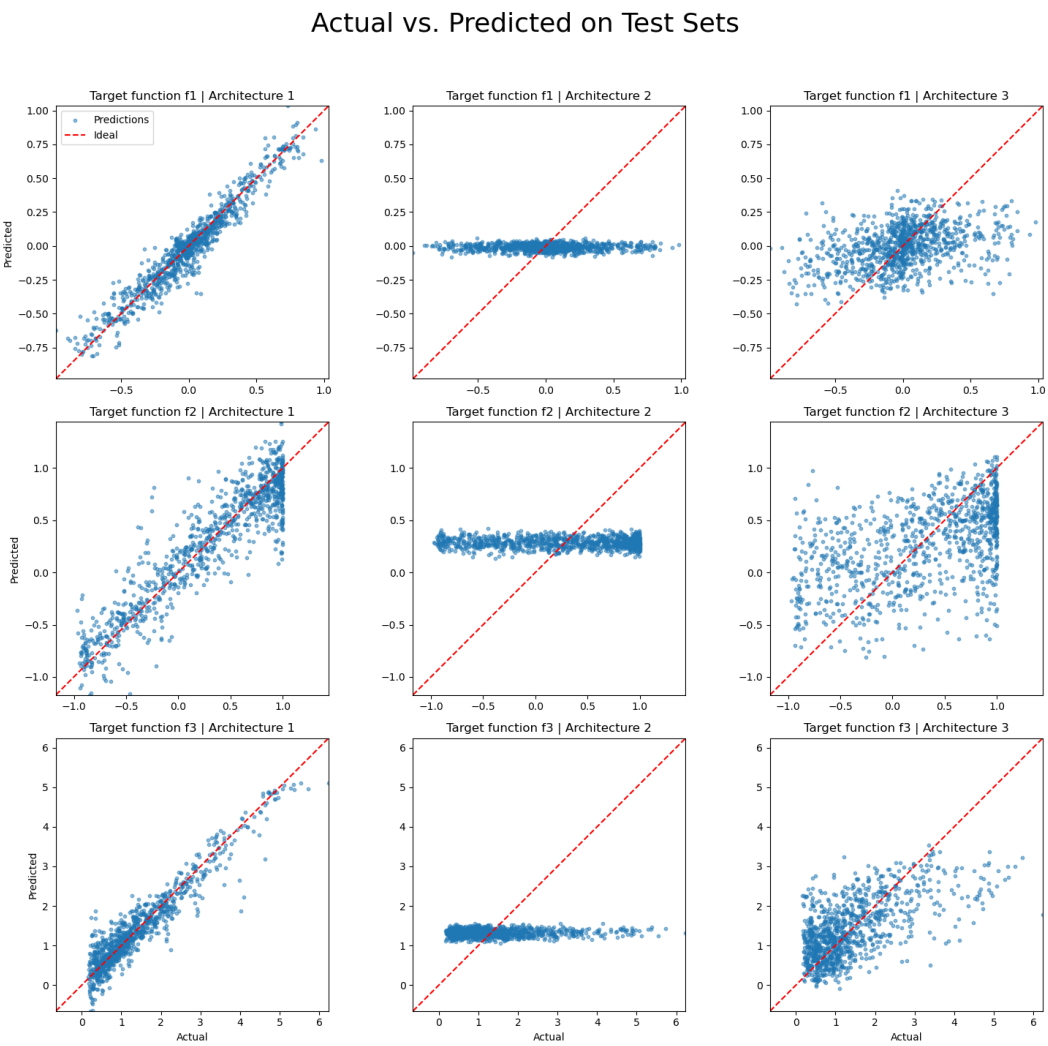


Figure 9: Actual vs. Predicted function values for each architecture-function pair. Architecture 1 consistently outperforms Architectures 2 and 3 across all functions.

related to this project, preliminary unpublished results show that the approximation error is proportional to the number of paths from input to output. Hence, for a fixed number of parameters, increasing the sparsity of the network provides a way to decrease the number of such paths, thereby decreasing the approximation error of the KAN.

8 Future Directions

While this analysis answers some questions, it leaves others unanswered. Most notably, we did not explore the relative performance of KANs vs. MLPs. Secondly, we intentionally confined the experiments to learning equations reminiscent of physical processes, such as those encountered in scientific settings, rather than tasks like image classification or next-word prediction, to maintain consistency with the existing literature on KANs, which has mainly focused on AI for science.

Anticipated future work falls along two axes: theoretical and experimental. In light of our forthcoming results linking sparsity to KAN's approximation error, a result that provides lower limits on the benefits of sparsity is desirable. On the experimental front, further work is needed to examine whether our results generalize to larger networks and other tasks.

References

[1] A. N. Kolmogorov and V. I. Arnold, *Representation of Multidimensional Continuous Functions as Superpositions of Univariate Functions*, Moscow University Press, 1957.

[2] Z. Liu, et al., *KAN: Kolmogorov-Arnold Networks*, *arXiv preprint arXiv:2404.19756*, 2024.

[3] J. Schmidt-Hieber, *The Kolmogorov-Arnold Representation Theorem Revisited*, *arXiv preprint arXiv:2007.15884*, 2021.

[4] D. Sprecher, *On the structure of continuous functions of several variables*, *Transactions of the American Mathematical Society, 1965*

[5] David A Sprecher and Sorin Draghici. Space-filling curves and kolmogorov superposition- based neural networks. Neural Networks, 15(1):57–67, 2002.

[6] Mario Köppen. On the training of a kolmogorov network. In Artificial Neural Networks—ICANN 2002: International Conference Madrid, Spain, August 28–30, 2002 Pro- ceedings 12, pages 474–479. Springer, 2002.

[7] Ji-Nan Lin and Rolf Unbehauen. On the realization of a kolmogorov network. Neural Com- putation, 5(1):18–20, 1993.

[8] Tomaso Poggio, Andrzej Banburski, and Qianli Liao. Theoretical issues in deep networks. Proceedings of the National Academy of Sciences, 117(48):30039–30045, 2020.

[9] Federico Girosi and Tomaso Poggio. Representation properties of networks: Kolmogorov's theorem is irrelevant. Neural Computation, 1(4):465–469, 1989.

[10] Quanta Magazine, Novel Architecture Makes Neural Networks More Understandable

[11] Li, Chenxin, et al. *U-kan makes strong backbone for medical image segmentation and generation,* arXiv preprint arXiv:2406.02918, 2024.

[12] Yueyang Cang and Yu hang liu and Li Shi, *Can KAN Work? Exploring the Potential of Kolmogorov-Arnold Networks in Computer Vision*,arxiv 2024

[13] app3 Minjong Cheon, *Demonstrating the Efficacy of Kolmogorov-Arnold Networks in Vision Tasks*, arxiv, 2024.

[14] Rigas, Spyros and Papachristou, Michalis and Papadopoulos, Theofilos and Anagnostopoulos, Fotios and Alexandridis, Georgios, *Adaptive Training of Grid-Dependent Physics-Informed Kolmogorov-Arnold Networks*, IEEE Access, 2024.

[15] Van Duy Tran and Tran Xuan Hieu Le and Thi Diem Tran and Hoai Luan Pham and Vu Trung Duong Le and Tuan Hai Vu and Van Tinh Nguyen and Yasuhiko Nakashima, *Exploring the Limitations of Kolmogorov-Arnold Networks in Classification: Insights to Software Training and Hardware Implementation*, arxiv, 2024.

[16] Ziming Liu and Pingchuan Ma and Yixuan Wang and Wojciech Matusik and Max Tegmark, *KAN 2.0: Kolmogorov-Arnold Networks Meet Science*, arxiv, 2024.

[17] Benjamin C. Koenig, Suyong Kim, Sili Deng, *KAN-ODEs: Kolmogorov–Arnold network ordinary differential equations for learning dynamical systems and hidden physics*, Computer Methods in Applied Mechanics and Engineering, 2024.