EITN35: Exploration of Kolmogorov-Arnold Networks for Hardware Implementation

Fuad Mammadzada,
Dr. Joachim Rodrigues, Masoud Nouripayam, Kristoffer Westring

January 28, 2025

Contents

1 Introduction

The exponential growth of data and the proliferation of interconnected devices have ushered in the era of Big Data and the Internet of Things (IoT). These developments have driven the need for advanced machine learning (ML) models capable of processing vast amounts of information in real time. Simultaneously, the rise of Large Language Models (LLMs) like GPT has underscored the computational demands of modern AI systems. These technologies, while transformative, pose significant challenges in terms of power consumption, scalability, and computation efficiency.

One of the widely used ML architectures is Multi-Layer Perceptron (MLP), where each layer of neurons is connected through linear weights and non-linear Activation Functions (AFs). All the linear calculations like multiplication and accumulation can be modeled as matrix multiplication. This approach enables the representation of MLPs using the extensive mathematical tools available from linear algebra. Due to their mathematical simplicity, scalability (the user can just increase the number of neurons and layers to reduce error), and inherent non-linearity, MLPs have been the backbone of today's ML & Artificial Intelligence (AI) research. They are the default model in ML for approximating non-linear functions, due to their power guaranteed by the Universal Approximation Theorem (UAT) [1].

Even though MLPs are used extensively, they have several significant drawbacks. First of all, MLPs suffer from large neural scaling factors—that is, the number of model parameters required to achieve a given reduction in test loss increases exponentially with the number of inputs to the model. This makes MLPs computationally expensive and memory-intensive, particularly for tasks involving high-dimensional data. In addition, MLPs are not inherently interpretable - it is not possible to get an insight into how the model works inside. For example, in the case of AlphaFold, they were able to train an ML model to accurately predict protein structures, but without fully understanding how the model performs the prediction. [2].

In their seminal paper named "KAN: Kolmogorov-Arnold Networks" [3], Ziming Liu et al. propose an alternative ML architecture inspired by Kolmogorov-Arnold representation Theorem (KAT). Like MLPs, KANs have fully connected structures, but utilize AFs directly on the edges between neurons, comparable to weights in MLPs. KANs have no linear matrices at all, each weight is replaced by a learnable 1D AF parametrized as a spline. Whereas MLPs have non-linear activations after the summation of edges in the node, KANs only sum the incoming signals from the edges without adding more non-linearities. One can think KANs are unnecessarily expensive since each weight in an MLP is a whole function in KAN, but by utilizing symbolic representations and spline-based AFs, they are able to achieve compact, interpretable architectures while maintaining high accuracy for various tasks, as shown in [3].

This project aims to investigate KANs for potential hardware implementation. The following is the list of tasks that are accomplished in this project with their corresponding sections in the report:

- The mathematical background of KANs: Section 2
- Train KAN for different datasets and compare it with MLP: Section 3
- Hyper-parameter analysis of KANs: Section 3
- Implement an inference behavioral model: Section 4
- Optimize the inference for hardware implementation: Section 4

2 Background

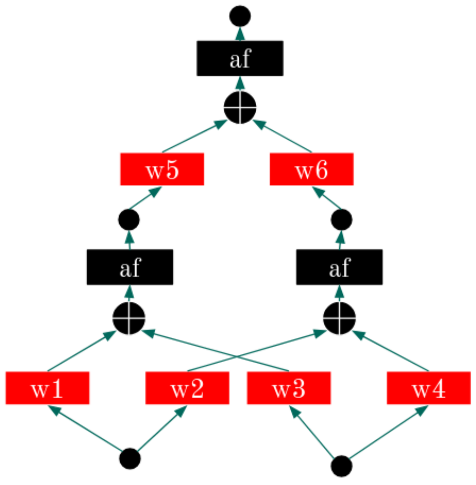
In this section, we provide the mathematical foundation of KANs, detailing their reliance on b-splines for constructing single-variable functions and their optimization through the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) algorithm. Furthermore, we explore existing hardware implementations of KANs in the literature, highlighting both their potential advantages and the challenges associated with efficient inference acceleration compared to traditional architectures like MLPs.

2.1 Mathematical foundation

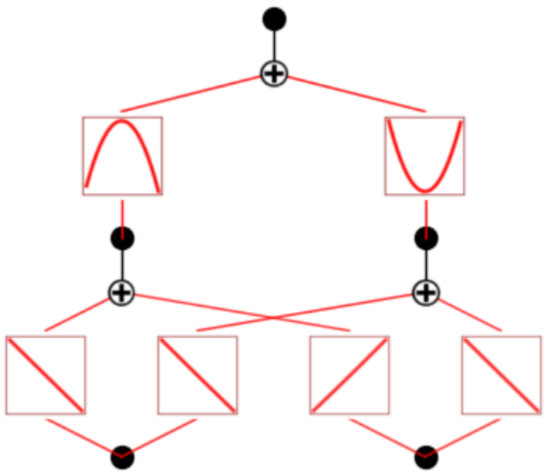
As mentioned in the Section 1, KANs are based on Kolmogorov-Arnold representation Theorem (KAT). This theory, proposed by Vladimir Arnold and Andrey Kolmogorov, established that if f is a multivariate function on a bounded domain, then f can be written as a finite composition of continuous functions of a single variable and the binary operation of addition. For smooth $f : [0,1]^n \rightarrow R$:

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

where $\phi_{q,p} : [0,1] \rightarrow R$ and $\Phi_q : R \rightarrow R$. What the Equation 1 entails is that for a given n variable input function f, we can dissect it as the summation and composition of single-dimensional functions. MLPs are just function approximators [1] and we can use KANs for the same task as well. In the Figure 1 below, the comparison between an MLP and a KAN architecture is given.



(a) MLP [2,2,1]



(b) KAN [2,2,1]

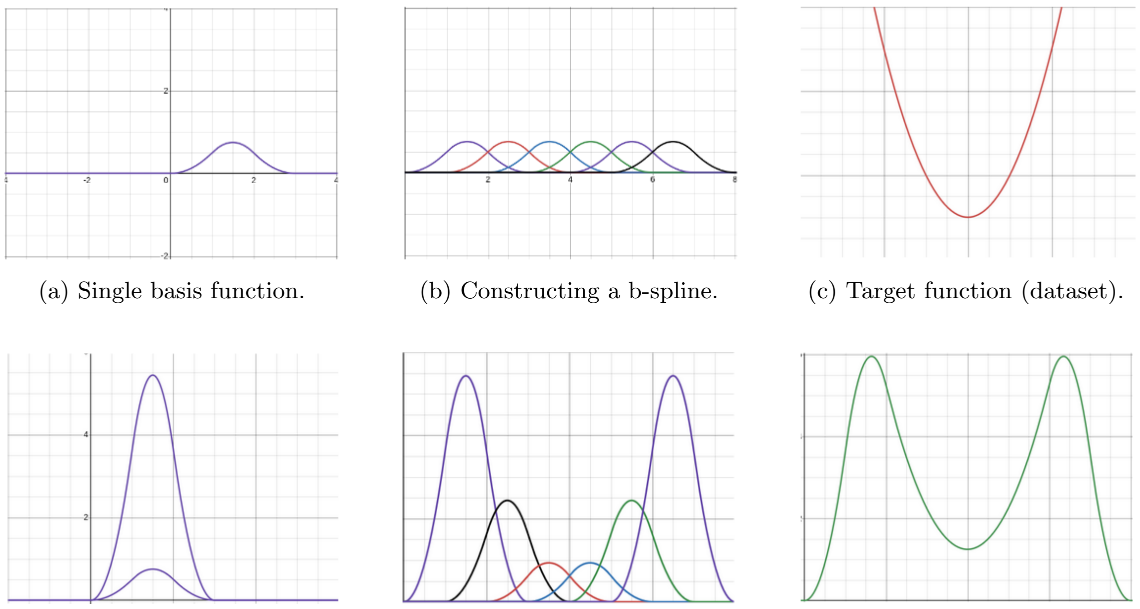
Figure 1: Comparison of KAN and MLP architectures

As we can see, in MLPs, inputs are multiplied by weights, summed on the outputs, and passed through given AFs. This process is different in KANs in the sense that we pass the inputs directly from AFs, which are trainable, and then sum in the outputs.

The main idea in the KAN paper is that they use a mathematical tool called b-splines to describe those single-dimensional functions. Splines are used to draw a function given a set of control points. They all connect several points in space more or less continuously, but different types of splines promise different mathematical properties for the generated functions [4]. B-splines, specifically, make sure that the resulting graph is second-degree continuous. This means each section of the spline is connected smoothly - that is the second-degree derivatives of each section are the same value. This property is important for training - as the KAN researchers use LBFGS optimizer to minimize the loss function. LBFGS is a quasi-Newtonian optimization algorithm that uses second-degree derivatives to calculate the ideal step direction for loss minimization. Using b-splines ensures that the KAN model is second-degree differentiable, making it suitable for LBFGS.

Visually speaking, b-splines are just summation of a set of phase-shifted, identical basis functions as can be seen in Figure 2 (a) and (b). The mathematical formulation of an individual basis function will be addressed after discussing the application of this tool for curve fitting or function generation. Assume that we want to fit the function (that is our dataset so to say) given in Figure 2 (c) using the set of basis functions depicted in (b). By carefully selecting a set of coefficients for each basis function, we can scale them as shown in Figure 2 (d) and (e). Then, by summing everything we can get a resulting approximation of the original curve depicted in (e). The main issue here is choosing the set of coefficients - it is an optimization or training problem.

Mathematically, we need to define the basis function in such a way that it ensures the property of second-degree differentiability. We use Cox-DeBoor's recursive algorithm to define such function [5]:



(a) Single basis function. (b) Constructing a b-spline. (c) Target function (dataset).

(d) Scaling a single basis. (e) Scaling of the whole grid. (f) Final b-spline fit.

Figure 2: Process of curve fitting using B-splines.

$$B_{i,0}(x) = \begin{cases} 1 & t_i \leq x < t_{i+1} \\ 0 & otherwise \end{cases} \tag{2}$$

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i,k-1}(x) \tag{3}$$

where, $B_{i,k}(x)$ is the value of k-order i^{th} basis-function evaluated at a given point x. Equation 2 defines the formula for 0th order basis function, which is then used recursively to build higher-order basis functions. We define the input interval to be divided by the G grid of knot intervals. The knot grid defines the intervals where these basis functions overlap. Each $[t_i, t_{i+1}]$ defines an interval, and we compare the input with these intervals to use the corresponding basis functions. After we've defined the formulas to calculate a single basis function, we can extend it to cover the whole b-spline with coefficients:

$$S(x) = \sum_{i=0}^{n} c_i B_{i,k}(x) \tag{4}$$

Here, we iterate over all the basis functions defined over each grid knot for a given x and multiply them with their corresponding coefficients. The procedure of generating a cubic ($k = 3$) b-spline can be explored further with the provided interactive graph in [6].

The authors of original KAN paper [3] go one step further and introduce residual AFs, similar to residual connections, such that the overall AF is the sum of residual AF and the spline AF:

$$\phi(x) = w_b b(x) + w_S S(x) \tag{5}$$

where $b(x)$ is the residual AF, and w_b and w_S are coefficients for residual and spline functions, respectively. In the original paper and in all our subsequent tests, SiLu has been used as the residual AF.

An important insight we can get from this discussion is that by increasing the number of basis functions, we can increase the accuracy of the approximation. For example, in Figure 2 (e), b-spline is able to fit the given curve to some degree, but after that, it just fails on the edges. To improve that we can simply increase the input interval and the number of knot grids, hence increasing the number of basis functions. The authors of the original paper [3] call this technique Grid Extension, and they use it often to increase the accuracy of the model arbitrarily without changing the given architecture. This is one of the advantages that KAN has over MLP - it can increase its accuracy in external (structural) as well as internal (grid extension) ways.

In addition, since KANs utilize learnable activation functions on the edges, these functions can be represented as symbolic expressions after training, making KANs well-suited for deriving human-understandable mathematical formulas or symbolic laws from data. In ML terms, this ability to understand and explain how a model makes decisions or predictions is called interpretability. The ability of KANs to symbolically represent the entire model or some parts of it is a significant advantage over MLPs, where such symbolic representation is inherently unfeasible [3]. The authors of the paper explores this extensively as well, calling the process Symbolification.

2.2 Hardware implementation

The computation of b-spline functions in KANs presents new challenges when it comes to hardware acceleration. The recursive nature of traditional B-spline evaluation, as described in Equations 2 and 3, complicates acceleration, especially for high-dimensional data and numerous control points.

Since the architecture is new, IC implementations in the literature are scarce. The first paper to discuss this possibility [7], compared the KAN and MLP for different classification datasets such as Wine, DryBeans, Mushroom and so on. The authors implemented the architectures in high-level synthesis (HLS) and ran the models on an FPGA. The findings revealed that while the classification accuracy of KANs is comparable to that of MLPs, their hardware efficiency is significantly lower. However, utilization of HLS, the lack of parallelization, and hardware-specific optimizations in this study led to poor resource utilization on the FPGA. They also didn't quantize the network and carried all the calculations in 32-bit floating-point representation. Despite the negative results, the study serves as a baseline for understanding the computational demands of KANs and highlights areas requiring improvement, such as memory access patterns and efficient coefficient computation. The authors also note that KANS require substantially longer training times than MLPs, ranging from 6.55x to 36.68x [7].

Table 1: Results for Dry Beans Dataset from paper [7].

Type	Metric	MLP	KAN
SW Training	Model Size	16,20,15,10,7	16,2,7
	Spline Info	N/A	G=6,k=3
	No. Params	892	414
	Accuracy	0.921	0.924
	Energy Time Product (ET)	0.37	520.56
	Power (W)	21.33	22.78
	PDP (W*s)	7.892	11353.5
HW Implementation	Frequency (MHz)	100	100
	Power (W)	0.67	14.802
	BRAMs	0	781
	DSPs	17	9111
	FFs	11328	734544
	LUTs	8894	1677558
	Latency (#Cycles)	835	1896
	Latency (Time) (ns)	8350	18960

Table 1 from paper [7] suggests that MLP beats KAN for Dry Beans dataset almost all software and hardware metrics. However, we can also see one point on which KANs perform better: number of parameters and the model shape. KAN learns the same dataset with less than half amount of parameters. If we could implement a more efficient inference model this promises us to get better results.

Table 2: Comparison of KAN and DNN ICs from paper [8].

Metric	**Reduction Achieved by KAN IC (Compared to DNN)**
Energy Consumption	51.04x to 77.97x
Area	9.28x to 41.78x
Latency	23.59x to 29.56x

Another paper, *Hardware Acceleration of Kolmogorov–Arnold Network (KAN) for Lightweight Edge Inference*, delves into hardware details, considering the possibility of using Look-Up Tables (LUTs) and Compute-In Memory (CIM) techniques to speed up the inference. Their work, on the other hand, is not tested exclusively for classification datasets [8]. They have shown that for the knot theory dataset, compared to a Deep Neural Network (DNN), their KAN IC achieves far better hardware efficiency, while achieving higher accuracy [8]. Their results are summarized in Table 2. These advancements are attributed to architectural optimizations that exploit the repetitive structure of b-spline evaluations, enabling parallel computation and efficient resource utilization.

As a summary of this section, we can write the following points as the advantages of KAN:

- Interpretable.
- Less parameters for the the same test loss.
- Very good results on mathematical tasks (e.g. PDEs, regression, different theories).
- second-degree continuity helps with LBFGS.

Disadvantages:

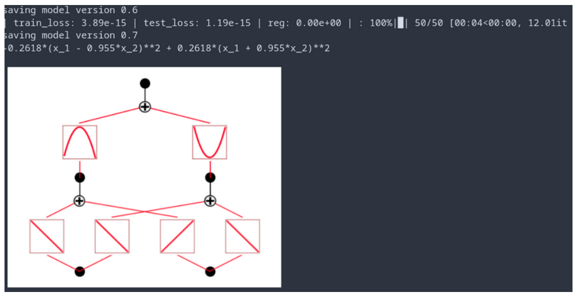
- Current results for classification tasks don't look promising in terms of hardware efficiency. According to paper [7], we can see that even though KANs reach similar accuracy with fewer number of parameters, their implementation in hardware can have a huge overhead. However, as noted this can potentially be resolved by looking into KANs from specifically hardware point-of-view.
- Slow training [7].
- KANs are more complicated to do inference. MLPs, on the other hand, utilize matrix multiplications, which have been optimized tremendously.

3 Training and Comparison with MLP

KAN researchers have developed a Python library [9] to be used for interacting with KAN models. The library is built upon PyTorch and is documented well enough for our exploration. I used this library throughout the project extensively, to train the models as well as understand how KANs work underneath.

3.1 Training for a toy dataset

As a first task, we sought to train a KAN to approximate the xy function, with the goal of both deriving an accurate symbolic representation and evaluating the performance of different optimization methods. Since KAN is based on mathematical functions, the PyKAN library provides means to try to fit well-known functions (squared, cubic, sine, log, etc.) after training with b-splines. This can potentially increase the accuracy while decreasing the number of parameters tremendously. In KAN literature this process is named Symbolification. With xy dataset we tried to achieve something similar as well. After training, the resulting network was ultimately symbolified as can be seen in Figure 3. If we try to simplify the resulting equation, we get 1.000076xy which is extremely close to the expected symbolic representation. This demonstrates that the neural network effectively learned the target function under optimized conditions.



```
saving model version 0.6
| train_loss: 3.89e-15 | test_loss: 1.19e-15 | reg: 0.00e+00 | : 100%|█| 50/50 [00:04<00:00, 12.01it
saving model version 0.7
-0.2618*(x_1 - 0.955*x_2)**2 + 0.2618*(x_1 + 0.955*x_2)**2
```

Figure 3: KAN trained for xy dataset.

Two optimization methods were evaluated for training the network: LBFGS and Adam. LBFGS, a quasi-Newton method solver that utilizes second-order derivatives, completed the training process in approximately 114 seconds and delivered highly accurate results. In contrast, Adam, a first-order optimizer based on Stochastic Gradient Descent (SGD), completed training faster, requiring only 90 seconds. However, the results obtained with Adam were significantly less accurate, indicating potential tuning issues. While Adam's efficiency can be attributed to its reliance on first-order derivatives, its performance in this case suggests that adjustments such as tuning learning rates or regularization parameters are required to achieve satisfactory accuracy.

During the training process, a major issue encountered was related to refinement levels in the grid size. The xy function failed to train successfully using the template training code provided by the PyKAN library, which was effective for other functions. Initial attempts to resolve this involved grid extension; however, excessively large changes to the grid size disrupted the network's ability to optimize properly. After iterative adjustments, a ”sweet spot” for the refinement level was identified, enabling successful training. This finding highlights the sensitivity of the training process to abrupt changes in the network's grid size and emphasizes the need for careful control of refinement parameters.

This experiment provided several key insights:

1. The effectiveness of LBFGS demonstrates the value of leveraging second-order optimization methods for b-spline parametrized AFs.
2. The challenges with refinement levels underscored the importance of gradual, controlled adjustments during training. Therefore, this is examined even further in the Section 3.3.
3. While Adam showed potential for faster training, further investigation is required to make it a viable alternative in terms of accuracy for this specific use case.

3.2 Comparison with MLP

After training KAN, we also created an MLP model for the same dataset to compare their accuracy. For the experiment, we utilized MLP models of 2 different structures: [2,39,1] and [2, 60, 1]. This is done to mimic the grid extension procedure we utilize in KAN to increase its accuracy. The results are summarized in Table 3.

The first set of results shows that as the number of parameters increases, the test loss decreases for both MLP and KAN, indicating that the models are becoming more accurate with more complex architectures. However, the KAN seems to exhibit a more dramatic improvement in test loss as the number of parameters increases, suggesting that it may be more efficient in utilizing its parameters compared to the MLP.

For the MLP, at 200 optimization steps, the test loss is 2.18e-04 with 157 parameters, and as the number of parameters increases to 241, the test loss improves to 1.31e-04. This indicates that increasing the model's complexity results in a better fit to the data, but the reduction in test loss is not as sharp, suggesting diminishing returns after a certain point in parameter expansion.

Table 3: Comparison of test loss for MLP and KAN with varying number of parameters for xy dataset.

Architecture	# of parameters	Test loss
MLP	157	2.18×10^{-4}
	241	1.31×10^{-4}
KAN	102	1.76×10^{-4}
	122	2.23×10^{-5}
	After symbolification	1.19×10^{-15}

In contrast, the KAN shows a more pronounced improvement. With 150 optimization steps, the test loss starts at 1.76e-04 for 102 parameters and decreases significantly to 2.23e-05 for 122 parameters. This is a significant reduction, and it seems that even a relatively small increase in the number of parameters leads to a dramatic improvement in the model's performance. The sharp decrease in test loss, especially compared to the MLP, points to the potential efficiency of KANs in learning from limited parameters.

An important discussion should be made regarding how the increase in parameter size was achieved. In the case of MLP, we created 2 separate models and they were trained independently. However, with KANs, we can use the grid extension technique simply to increase the number of parameters, without losing the previously trained model. The new, refined model is then trained based on the previous one, saving us a lot of time between these micro-improvements. **In summary, improving MLPs requires modifying the entire architecture, whereas KANs can be enhanced through grid extension without discarding prior training.**

Finally, after symbolification, the test loss for the KAN drops to an extraordinary 1.19e-15. This represents a nearly perfect fit, which highlights the potential of KANs for extremely high levels of model compression or accuracy when applied in hardware or resource-constrained environments. The fact that symbolification further improves the test loss underscores that KANs are particularly well-suited for datasets with a clear mathematical foundation, as symbolic optimization techniques can drastically reduce computational complexity while maintaining or even enhancing model performance.

These results imply that KANs are not only more efficient than MLPs in terms of parameter usage but also that they have the potential for much higher accuracy, especially when optimized through symbolic methods. This can be particularly advantageous in hardware implementations where parameter size and computation are critical constraints. **However, we should be cautious about our findings as well, since this is only a simple function regression task, with a harder classification task, it can be challenging to fit generated b-spline AFs into symbolic functions.**

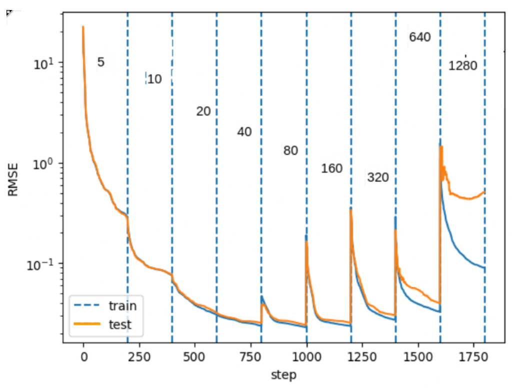
3.3 Hyper-parameter Analysis

The main hyper-parameter that is in our discussion is the grid size. We set-up two experiments to see the effects of increasing grid size on the training and testing loss of the KAN. First, we run the KAN training for 2000 steps and after each 200 steps we double the grid size while continuing the training. This experiment pushed the grid size as high as 1280 intervals which caused the training to become extremely slow. The second experiment, on the other hand, increases the grid size with linear steps of 10 after each 100 steps.

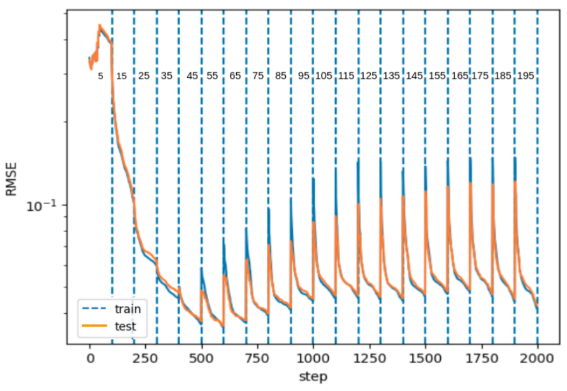
The graphs depicted in Figure 4 summarizes the results generated from these experiments. We analyze the performance of grid extension, with Root Mean Square Error (RMSE) as the primary evaluation metric for both training and testing datasets.

A side-by-side comparison of the two strategies reveals key differences in their behavior and applicability:

1. **Adaptability to Grid Changes**: The system adapts more effectively to smaller, incremental grid changes under the linear strategy. This is evidenced by the smaller and more



(a) Exponentially increasing grid extension.



(b) Linearly increasing grid extension.

Figure 4: Grid extension experiments for xy dataset.

predictable RMSE fluctuations compared to the exponential approach.

2. **Trade-offs Between Speed and Stability**: The exponential strategy allows for faster exploration of larger grid sizes, achieving lower RMSE at certain stages. However, it comes at the cost of higher instability after each extension. Conversely, the linear strategy is slower but ensures a steady and stable learning process.

3. **Efficiency of Stabilization**: While the exponential strategy exhibits significant RMSE spikes, the recovery period tends to increase as the grid size grows larger. The linear strategy, by contrast, maintains a smoother progression with minimal disruptions, ensuring consistent improvement throughout.

4 Inference Behavioral Model

This section explores how we built an inference behavioral model to be used for hardware design. The Inference Model (IM) is an essential step in creating an optimized hardware for KANs, as we can see all the algorithmic-level optimizations that are available for implementation. This will enable us to co-design algorithms and hardware for maximum optimization.

The general steps for the inference of KAN is following:

1. **Grid and Coefficient Retrieval.** For each layer, the corresponding grid points, coefficients, and scaling factors are extracted from the state dictionary.

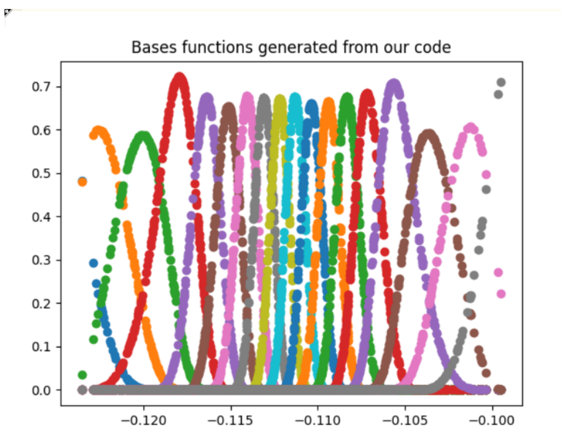
2. **Evaluate Basis Functions.** The basis functions are evaluated: Step functions are initialized based on the input's position relative to the grid (Equation 2). Higher-order basis functions are recursively computed using the knot points based on Equation 3.

3. **Scale Basis Functions.** Each basis function is multiplied by its corresponding coefficient, and the weighted outputs are summed to produce the spline output.

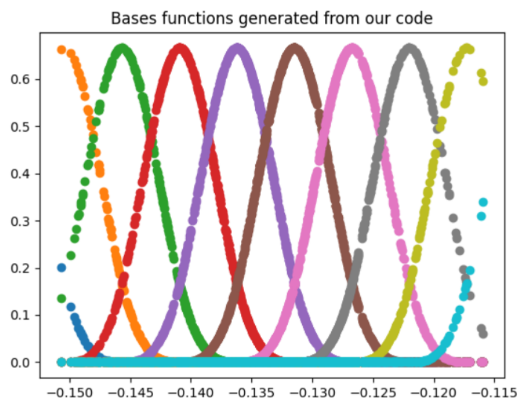
4. **Apply SiLU Activation for the Residual Pass of Inputs.** A SiLU AF is applied to the input values and scaled by the parameter.

5. **Combine Outputs.** The spline output and the SiLU output are combined to produce the layer's final output.

Based on the Equations 2, 3, and 4, we implemented a KAN IM in Python, utilizing tensors from PyTorch as the main data structure. This implementation revealed an important insight: after training, the grid is not distributed uniformly but is instead arranged strategically, allocating more basis functions to input ranges where the data is more concentrated or prevalent. This fact is depicted in Figure 5. If we cannot achieve some degree of uniformity among the basis functions associated with the same AFs, evaluating the corresponding equations for each grid interval will become highly inefficient for hardware implementation. Uniformity allows identical basis functions



(a) Non-uniformly distributed.



(b) Uniformly distributed.

Figure 5: KAN[2,2,1] basis functions.

for each AF, enabling optimizations through a general equation or shared LUT [8], which is not possible without it.

To address this issue, we shifted our focus from hardware design back to the KAN software training process. Further exploration of the PyKAN library revealed that the KAN object accepts a parameter called ϵ, which controls the degree of uniformity between the grid intervals. By setting this parameter to 1, we ensured that the KAN library maintains uniformly spaced grid intervals during training, effectively achieving the desired level of uniformity. After setting this parameter, we got the basis functions as depicted in Figure 5 (b).

After resolving the uniformity issue, we proceeded with the inference model and empirically calculated the number of different types of calculations required for a single forward pass of the input through the whole network. These computations are categorized into two sources:

- **Basis Generation**: evaluating the value of *each basis function* corresponding to the given input for the AF. AFs connected to the same input nodes share the same basis functions, since the input range and grid interval are the same.
- **Layer Evaluation**: the basis function values are multiplied by corresponding coefficients and summed to construct the actual AF.

Table 4: Computation breakdown for the initial inference model.

	Basis Generation	**Layer Evaluation**	**Total**
Additions [2,2,1]	1452 (95%)	78 (5%)	1530
Multiplications [2,2,1]	264 (78.5%)	72 (21.5%)	336
Divisions [2,2,1]	264 (97.7%)	6 (2.3%)	270
Additions [2,20,20,1]	15246 (72%)	5980 (28%)	21226
Multiplications [2,20,20,1]	2772 (33%)	5520 (67%)	8292
Divisions [2,20,20,1]	2772 (86%)	460 (14%)	3232

This analysis allowed us to identify the primary bottleneck in the inference process of KAN: the majority of computations are spent on generating the basis functions. To optimize the model, we focused on reducing the complexity of this step. The following procedure was implemented to address this issue. First, we fixed the order of the B-spline to a predefined value: k=3. In all our previous tests, we used cubic splines, and the literature suggests that increasing the spline order does not yield significant performance improvements [3]. Consequently, by reducing the analysis of the project to cubic splines, a closed-form solution was found, resulting in a significantly improved computational cost. The derivation for the expression found in this project is shown below.

$$b_{i,3,1}\left(x\right) = \frac{\left(x - t_i\right)^3}{6u^3} \tag{6}$$

$$b_{i,3,2}\left(x\right) = \frac{\left(x - t_i\right) \cdot \left(\left(x - t_i\right) \cdot \left(t_{i+2} - x\right) + \left(t_{i+3} - x\right)\left(x - t_{i+1}\right)\right) + \left(t_{i+4} - x\right) \cdot \left(x - t_{i+1}\right)^2}{6u^3} \tag{7}$$

$$B_{i,3}\left(x\right) = \begin{cases} b_{i,3,1}(x) & t_{i+1} > x \geq t_i \\ b_{i,3,2}(x) & t_{i+2} > x \geq t_{i+1} \\ b_{i,3,2}(t_i + t_{i+4} - x) & t_{i+3} > x \geq t_{i+2} \\ b_{i,3,1}(t_i + t_{i+4} - x) & t_{i+4} > x \geq t_{i+3} \\ 0 & otherwise \end{cases} \tag{8}$$

where, t_i is the knot in the grid, and u is the grid interval defined as the difference between two consequent knots. From the equations above, we can observe two key points. Firstly, a cubic b-spline consists of four pieces, defined over four grid intervals. Secondly, we do not actually require four distinct functions to define the basis function; instead, we use only two. This simplification is made possible by our prior discussion on uniformity. Having a uniform grid ensures that the basis functions are symmetric and identical. As a result, by mathematically defining half of the functions, the remaining half can be derived by carefully mirroring the first two.

During our research, we discovered another important feature of b-splines: *local control.* This property means that each point in the resulting **spline AF is influenced only by a small number of nearby basis functions**. Specifically, for cubic b-splines, assuming $t_i > x \geq t_{i+1}$, then AF value $S(x)$ depends only on $B_i(x)$, $B_{i-1}(x)$, $B_{i-2}(x)$, $B_{i-3}(x)$. In other words, only four basis functions are required to evaluate the AF output, rather than the entire grid of basis functions. Hence, the following modification can be made to the equation 4:

$$S(x) = \sum_{j=i-3}^{i} c_i B_{j,k}(x) \tag{9}$$

By leveraging this property, we implemented an optimized basis function evaluation that significantly reduces computational overhead. This approach enables an inference model whose performance (in terms of the number of computations) is independent of the grid size. Consequently, we can arbitrarily increase the model's accuracy by refining the grid without incurring any penalty on the inference hardware. The results are summarized in Table 5.

Table 5: Computation breakdown for the optimized inference model.

	Basis Generation	**Layer Evaluation**	**Total**
Additions [2,2,1]	144 (73%)	54 (27%)	198 (7.27x)
Multiplications [2,2,1]	68 (65%)	36 (35%)	104 (3.23x)
Divisions [2,2,1]	4 (40%)	6 (60%)	10 (27x)
Additions [2,20,20,1]	1512 (27%)	4140 (73%)	5652 (3.75x)
Multiplications [2,20,20,1]	714 (21%)	2760 (79%)	3474 (2.38x)
Divisions [2,20,20,1]	42 (8%)	460 (92%)	502 (6.438x)

Key insights:

1. The optimized design reduces computational costs across all operations, particularly for divisions, which are reduced by up to 27x in the smaller configuration.

2. Previously, the majority of division operations were primarily attributed to basis function generation. However, with the new design, this has shifted to layer evaluation, primarily due to the SiLU function used in the residual connection. This number can be further reduced

by replacing SiLU with a simpler AF, such as ReLU, which doesn't need divisions. In fact, all divisions can be bypassed, if we store the division terms in Equations 6, 7, and 8.

3. The decoupling of computational complexity from grid size and the spline order allows for scalable and efficient inference, even as the grid size increases for more complex networks.

5 Discussion on Hardware Implementation

Our analysis of hardware implementations revealed limitations in previous studies, including inefficient resource utilization and lack of optimization. By comparing KANs to MLPs, those authors have found that KANs required fewer parameters to achieve comparable results. This efficiency, combined with their interpretability, positions KANs as a viable choice for applications where computational and memory constraints are critical.

In developing the inference behavioral model, we identified that the majority of computational costs stemmed from basis function generation. By deriving closed-form equations for cubic B-splines and capitalizing on their local control property, we were able to optimize the inference process, reducing computational complexity significantly. The optimized inference model demonstrated that KAN performance is independent of grid size, making it scalable and efficient, even for larger networks. Additionally, we propose that replacing computationally expensive SiLU with simpler alternatives such as ReLU could eliminate divisions and enhance efficiency, though the impact on accuracy hasn't been explored.

A promising avenue for future research is the application of quantization techniques. By reducing the precision of calculations, quantization can significantly decrease hardware resource usage, power consumption, and latency without substantial impact on the result accuracy. This approach has been widely adopted in traditional NNs and could address the high computational demands of KANs, particularly during inference. Exploring quantization, along with other hardware-specific optimizations such as parallelization, could pave the way for more efficient and scalable KAN implementations in resource-constrained environments.

6 Conclusion

This project explored Kolmogorov-Arnold Networks (KANs) as a novel alternative to traditional Multi-Layer Perceptrons (MLPs) for machine learning tasks, with a focus on their potential for hardware implementation. By delving into the mathematical foundation of KANs, rooted in the Kolmogorov-Arnold Representation Theorem, we highlighted their unique architecture, which replaces linear weights with spline-based AFs. This design not only makes KANs interpretable but also introduces computational challenges that we addressed throughout the project.

Through training experiments, we demonstrated the ability of KANs to approximate complex functions with high accuracy. The process of symbolification further emphasized the network's capability to extract symbolic representations, drastically improving accuracy and interpretability while reducing computational demands. KANs showed significant improvements in test loss compared to MLPs, especially when leveraging grid extension techniques. These experiments also revealed the importance of controlled and incremental grid refinement to maintain stability during training.

Despite the promising results, challenges remain. Training KANs can be slow and sensitive to hyperparameter settings, and their potential for classification tasks requires further exploration. Future work could focus on optimizing training processes, exploring hardware-specific innovations like quantization and compute-in-memory techniques, and extending KAN applications beyond regression tasks to more classification domains.

In conclusion, KANs offer a compelling alternative to MLPs, blending interpretability with computational efficiency. By addressing their current limitations and continuing to refine their design, KANs could play a pivotal role in bridging the gap between high-performance machine learning and efficient hardware implementation, paving the way for advancements in modern AI systems.

References

[1] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[2] J. Jumper, R. Evans, A. Pritzel, *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 583–589, 2021.

[3] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T. Y. Hou, and M. Tegmark, "KAN: Kolmogorov-Arnold Networks," *arXiv preprint arXiv:2404.19756*, 2024.

[4] F. Holmér, "The Continuity of Splines." https://www.youtube.com/watch?v=jvPPXbo87ds, 2022. Accessed: 2024-12-12.

[5] C. de Boor, "Subroutine package for calculating with B-splines," Technical Report LA-4728-MS, Los Alamos Scientific Laboratory, Los Alamos, NM, 1971.

[6] A. Singh and F. Mammadzada, "Interactive Graph: B-Splines Visualization." https://www.desmos.com/calculator/bn3jril1xm. Accessed: 2024-12-12.

[7] V. D. Tran, T. X. H. Le, T. D. Tran, H. L. Pham, V. T. D. Le, T. H. Vu, V. T. Nguyen, and Y. Nakashima, "Exploring the Limitations of Kolmogorov-Arnold Networks in Classification: Insights to Software Training and Hardware Implementation," 2024.

[8] W.-H. Huang, J. Jia, Y. Kong, F. Waqar, T.-H. Wen, M.-F. Chang, and S. Yu, "Hardware Acceleration of Kolmogorov-Arnold Network (KAN) for Lightweight Edge Inference," 2024.

[9] Z. Liu, "PyKAN: A Python Library for Kolmogorov–Arnold Networks." https://github.com/KindXiaoming/pykan, 2024. Accessed: 2024-12-12.