A COMPREHENSIVE STUDY OF KOLMOGOROV ARNOLD NETWORKS

Deep Learning 2023 - 2024

Iria Quintero García
Javier González Otero
Arol Garcia Rodríguez

Contents

Project description

Our project studies the potential of Kolmogorov-Arnold Networks (KANs), a cutting-edge deep learning architecture inspired by the Kolmogorov-Arnold representation theorem. This architecture was published ensuring that it could offer significant improvements over traditional neural networks in various tasks. The aim of our research was to thoroughly evaluate the capabilities and applications of KANs in both theoretical and practical scenarios.

Initially, we replicated several foundational experiments from the original KAN research to validate our understanding and to ensure our setup was correctly implemented. Following this, we extended these experiments to explore new applications and configurations, aiming to push the boundaries of what KANs can achieve.

The main activities and experiments included in our work are the following:

1. **Function Fitting:** We tested KANs on different mathematical functions to evaluate their performance in accurately modeling and predicting the dataset's structure. This involved observing how well KANs generalize from known and unseen data from complex datasets.
2. **Grid Size and Optimizer Testing:** We experimented with different grid sizes and spline degrees to understand their impact on the performance of KANs. Additionally, we compared various optimization algorithms, such as Adam and LBFGS, to determine the most effective methods for training KANs.
3. **KAN Architectures:** By analyzing different architectures of KANs, we aimed to identify the most efficient configurations. This involved trying different widths of the architecture, different numbers of neurons per layer, and other parameters to optimize the performance for specific tasks.
4. **Real-world Applications:** We applied KANs to real-world datasets to test their practical utility. We used KANs for binary classification with a heart disease dataset, analyzing their accuracy and robustness in medical data prediction. Furthermore, we integrated KAN layers with classical Convolutional Neural Networks (CNNs) to enhance image recognition tasks, using datasets like SVHN and MNIST to benchmark performance.

Throughout the project, we documented each experiment, ensuring that our methods are reproducible.

A notebook for the different function fitting experiments is included in the project repository together with two other notebooks for the binary classification task in the heart disease dataset and for analyzing the performance with CNNs.

State of the art

Kolmogorov-Arnold networks have been recently introduced to the deep learning sphere. In fact, it has not even been a month since the original paper [1] release. However, KANs have already become an object of interest. Many people state that they are the foundation of *Machine Learning 2.0* and will replace MLPs.

Authors of the original paper [1] promise that KANs constitute an alternative to multilayer perceptrons that provides higher interpretability and efficiency.

What are KANs?

KANs are a new type of Deep Neural Network. Their main difference with respect to previous machine learning algorithms is that they are constituted by learnable activation functions. They are inspired by the *Kolmogorov Arnold representation Theorem,* which states that if a multivariate function $f(x_1, \dots, x_n)$ is a continuous 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.

Specifically:

$$f(X) = f(x_1, \dots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right) \quad (1)$$

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

Contrary, MLPs have foundations in the well known universal approximation theorem. Figure 1 shows a comparison between a regular MLP and the KAN approach.

KAN networks replace weights in edges (which from now on will be our neurons) by univariate functions, which are parametrized as *B-splines*. KANs have no linear weights at all. All these non-linearities are summed up on nodes, following the *Kolmogorov Arnold theorem*.

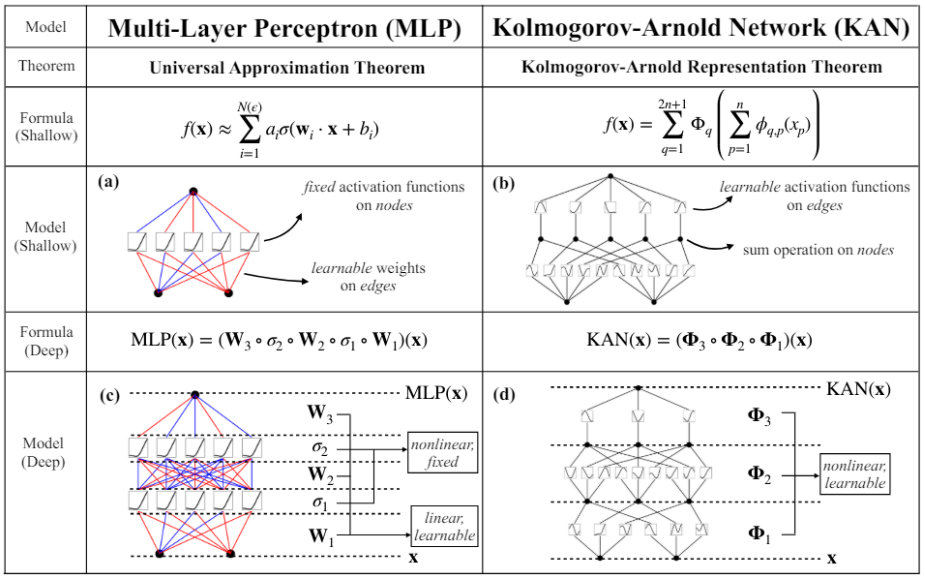


Figure 1: MLPs architecture vs KAN architecture (Figure provided by original paper)

As visible in Eq. (1), non linearities in the original equation have just *2n* levels of depth. However, the authors decided to not stick to that exact equation but add an unlimited number of layers. Figure 2 shows a more detailed example of the KAN network.

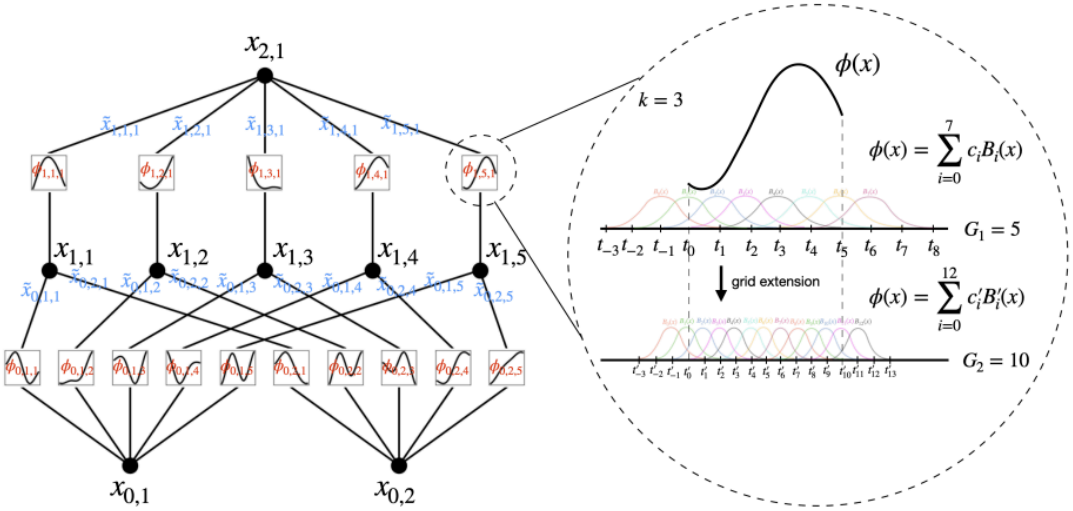


Figure 2: KAN network

Understanding the network

Anatomy of nonlinearities

At first glance, the left side of Figure 2 may seem a bit confusing. Let's explain how these nonlinearities are formed.

Splines

How could one learn a function? Bézier curves come to rescue us here, in concrete, through B-splines. B-splines are a combination of stacked Bézier curves of the same order/degree, making a twice differentiable (C^2 continuity) and with a smooth curvature (G^2 continuity) spline. Equation 2 shows the formulation of a B-Spline.

$$\phi(x) = \sum_{i=0}^{n} c_i B_i(x) \quad x \in [0, 1] \quad (2)$$

Where n is the number of control points, c_i are the **control points, WHICH ARE THE ONLY LEARNABLE PARAMETER** and $B_i(x)$ are the basis functions, which can be thought of as the weight of each learning point for a specific domain value of x. Basis functions will be better covered later in this section.

Degree

B-splines need a degree parameter to be defined. The degree of these curves is just the number of control points that define every Bézier curve that is stacked to form up the whole B-spline. Or in a more technical way, the maximum number of Basis functions that interact with a single domain value.

Basis functions

Although the authors do not explain this concept in their paper, we consider it important to be able to understand everything well.
Basis functions are constructed through the *Cox-de-Boor* recursion.

$$B_{i,0}(x) := \begin{cases} 1 & \text{if } t_i \leq x < t_{i+1}, \\ 0 & \text{otherwise.} \end{cases}$$

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

Figure 3: Construction of a Basis function through Cox-de-Boor

In Figure 3, index i represents the same thing as for equation 2, so, the control point for which the basis function is multiplied, and k is the degree of the curve, therefore, a Basis function for a B-spline of degree 3 will be defined through a recursion of depth 3.
Figure 4 [7] shows a set of Basis functions in different colors. For every control point of the curve s its weight is defined as a linear combination of the basis functions value in the same domain point.

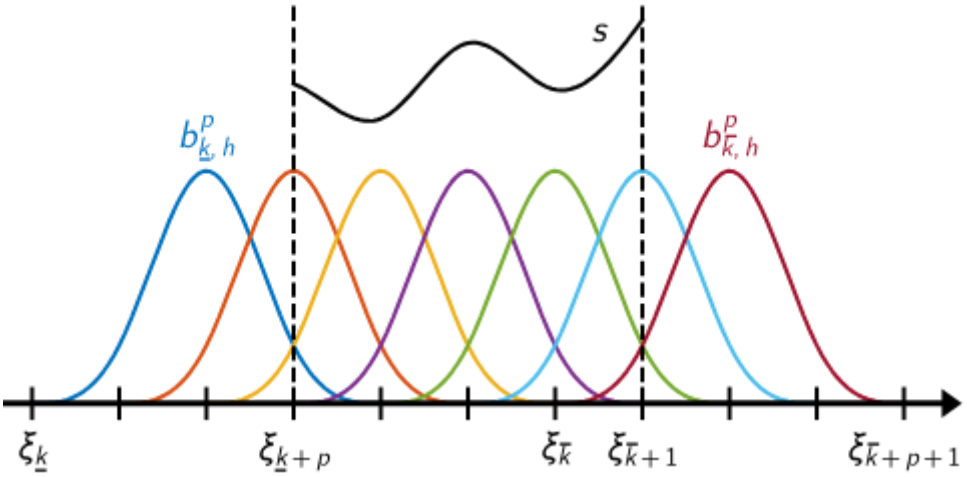


Figure 4: Basis functions (in colors) defining a spline (s)

Grid

This element of B-splines will be widely used during this work. It is important to define it exactly and learn how to deal with it.
The grid size, G, represents the number of stacked Bézier curves that form every B-spline. Therefore, they also define exactly the number of control points of the curve, being $G + k = number\ of\ control\ points$. For example, in Figure 2 left, we have 8 control points and a grid size of 5 with curves of degree 3.

Residual activation functions

For implementation purposes, authors decided to form nonlinearities as a linear combination of previously explained B - splines and a regular non linearity function (typically a Silu) acting as a residual connection. Equation 3 shows the final representation of an activation function.

$$\phi(x) = w_b silu(x) + w_s spline(x) \quad (3)$$

Authors do not explain a concrete reason behind this apart from being an optimization tool. We believe that this could be a font of future work.

Combining everything

To set up a KAN network, we need to order previous nonlinearities.

KAN Layer

Basically, what authors perform is to set some number of activation functions to the inputs, for instance, for input x_1 we may set [1, …, n] activation functions which take x_1 as input, and for x_2 we set the same n number of activation functions that take x_2 as input. Then, they are summed in order, for instance, result of first activation function of x_1 is summed with result of first activation function of x_2 and so on. This can be clearly visualized in Figure 2 (left).

KAN Network

The result of previous summations serve as input for the next KAN Layer. One can stack as many layers as needed.

Nomenclature

For coding purposes, the nomenclature used in authors implementation to form a KAN network is:

KAN(width=[2, 2, 1], grid=3, k=3)

Which refers to a KAN network of 2 input dimensions, 2 activation functions to every dimension, and an output of 1 dimension which combines by summing the output of the previous layer. With grid size 3 and order of Bézier curves 3. Figure 5 shows the architecture of this network.

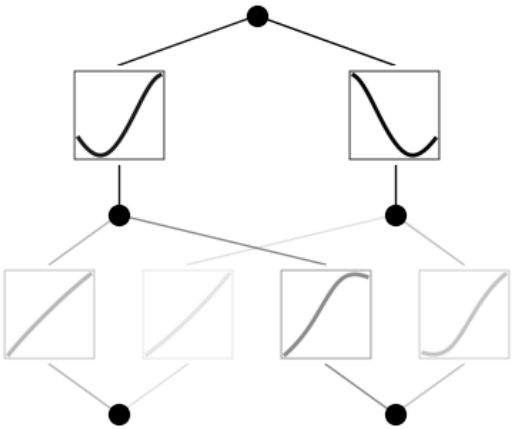


Figure 5: KAN 2, 2, 1 g=3, k = 3

Figure 5 network has been trained, so as noticeable, there already exist some nonlinearities.

Number of parameters

Authors provide an approximation to the number of parameters of the network, however, we decided to provide a formula and code to calculate the exact number of parameters.

As highlighted before, the only learnable parameters are control points of every activation function. For that reason we first need to know how many nonlinearities the network has. This number coincides with the number of parameters of an MLP since we have an activation function for every "edge" of a traditional MLP, therefore, for a KAN of *n* layers, the number of activation functions is obtained as in Equation 4.

$$\#activation = \sum_{i=1}^{i=n-1} width_i * width_{i+1} \quad (4)$$

Where $width_i$ is the size of the layer i.

Now, for every activation function, we have a fixed number of control points which, as one may expect due to previous points, depend on the grid size and the order of splines. Equation 5 explains the number of control points of every activation function.

$$\#Control\ per\ activation = Grid\ size + order\ of\ splines \quad (5)$$

Therefore, combining everything up, we know that the number of parameters is as in equation 6.

$$\#parameters = (Grid\ size + order\ of\ splines) * \left[\sum_{i=1}^{i=n-1} width_i * width_{i+1}\right] \quad (6)$$

We have provided a function to calculate this in the *function_fitting* notebook.

Function fitting

Authors have decided to try KANS on mathematical datasets. We decided to begin this project by learning how to use the *pykan* library, replicating some of their steps and adding some additional trials.

The main objective here is to perform a supervised learning task, in which our algorithms will try to predict the value of a function given one of its domain points. We will perform different experiments, some of them replicated from authors' examples [2], and some additional steps that involve some future work stated by authors as well as different demonstrations of the KAN library.

Every experiment here can be reproduced following the *function_fitting* notebook in this repository.

Basic KAN example

We will introduce this section by performing an example of mathematical data fitting through a KAN. Our simple model will learn the function $f(x, y) = x * y$, which by the Kolmogorov Arnold Theorem, we know that can be decomposed as $f(x, y) = \frac{1}{4}[(x + y)^2 - (x - y)^2]$ and therefore, represented by a [2, 2, 1] KAN in which functions of the first layer will correspond to linears and functions of the second layer will correspond to quadratics, the last layer is always an addition, therefore, no activation function is needed. The manifold of the dataset can be visualized in Figure 1. We have sampled some points and created training and test datasets. Training dataset can be visualized in Figure 2.

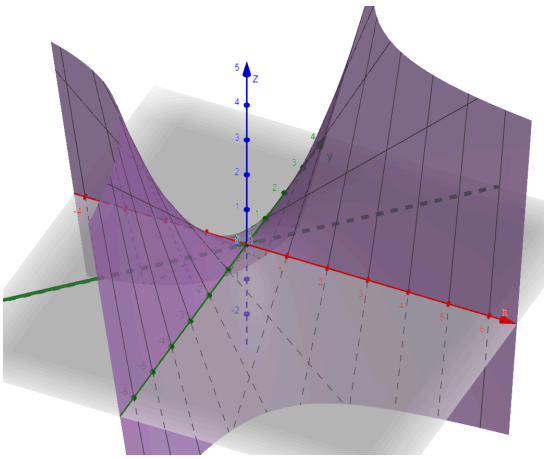


Figure 6: $f(x, y) = xy$

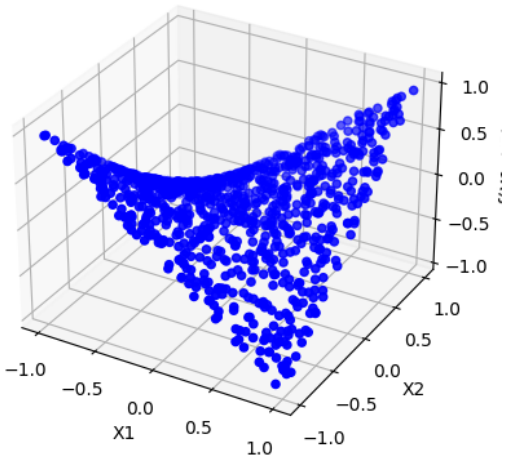


Figure 7: Training dataset

The train data and test data are composed of 1000 samples.

Since in this case we know a priori how the KAN should behave, we created a [2,2,1] KAN and as proposed by authors in *Example 1: Function fitting* [2] we trained it with grid size 3, B-splines degree of 3, *LBFGS* [3] optimizer and 20 steps. The number of parameters for this architecture is (2*2 + 2*1) * (3+3) = 36. Figure 3 shows the architecture design and Figure 4 shows the resulting KAN post training.

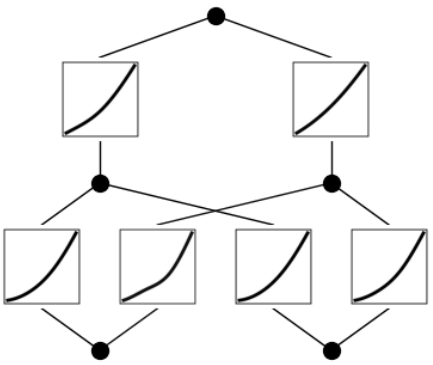


Figure 8: Pre-trained [2,2,1] KAN

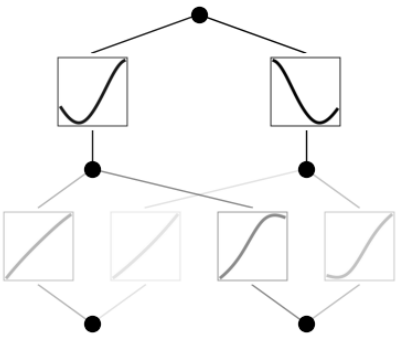


Figure 9: post training

One of the most valuable features of KAN models is their **interpretability**. In Figure 4, we can see clearly what our network has learnt, which is quite near to our expectations. Linearities for the first layer and somehow quadratics for the second layer. Despite this representation may not seem as accurate due to the lack of a negative linearity for the first layer, the model has managed to get a tiny last RMSE of 0.00401 within just 20 steps of *LBFGS*. The evolution of losses can be visualized in Figure 5.

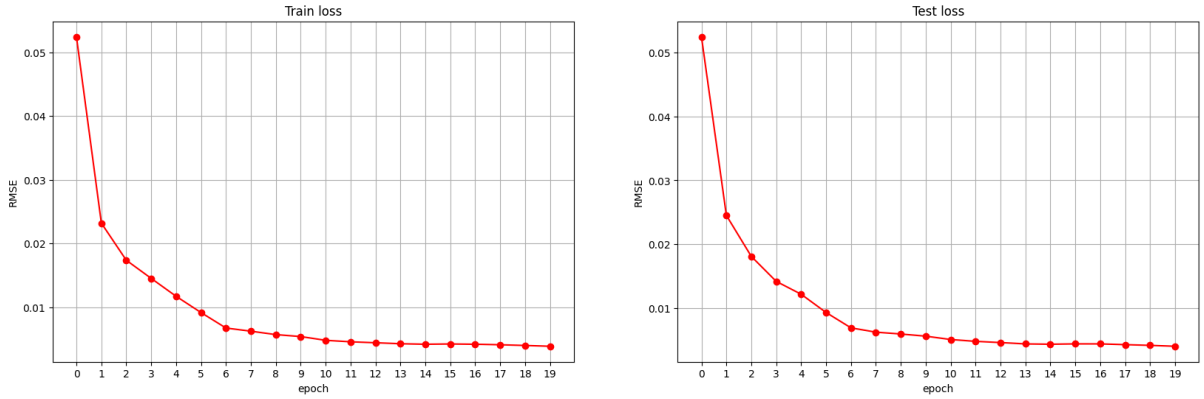


Figure 10: Training loss (left), Test loss (right). RMSE loss

For this example, training time was 6.47s . Output can be visualized in Figure 6.

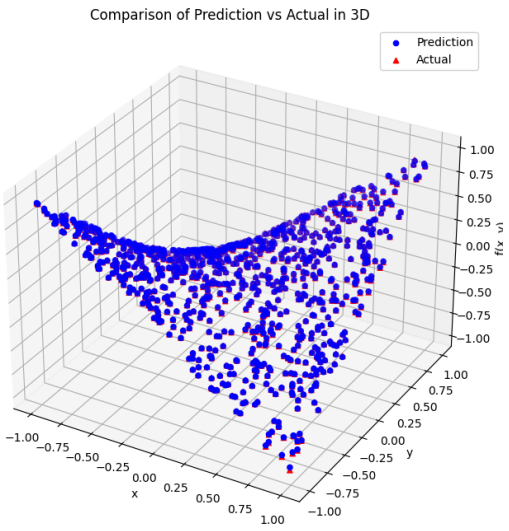


Figure 11: Predicted vs GT

Refining the grid size

Once seen the incredible results of previous KAN with a coarse grid of size 3, one may naturally ask for the effect of setting a finer grid size.

But what does it mean to refine the grid size? Why do we obtain a lower loss value by performing that technique? As previously stated, KAN activation functions are constructed as B-splines, therefore having a fixed number of control points within some closed domain. The grid size represents the number of splines that form the actual B-spline in a neuron, therefore, by augmenting them, we are also augmenting the number of control points, increasing the flexibility of B-splines to adapt to a more difficult scenario. This effect can be visualized in Figure 7.

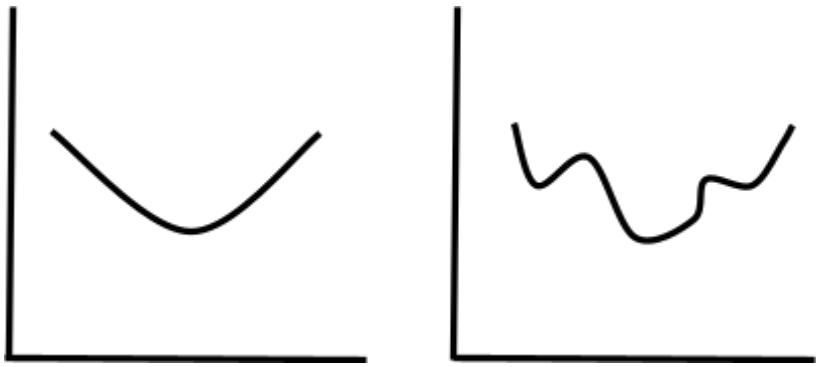


Figure 12: B-spline with reduced number of control points (left) vs big number of control points (right)

In *Example 1: Function fitting* [2], authors decided to initialize another model with grid size 10 from their model with grid size 3, performing a transfer learning technique. That led to an incredible reduction in loss for their dataset $f(x, y) = e^{sin(\pi x)+y^2}$. However, we decided not to perform such a procedure, instead, we directly trained a KAN with the same architecture as previous but grid size 10. Figure 8 shows RMSE losses for this experiment.

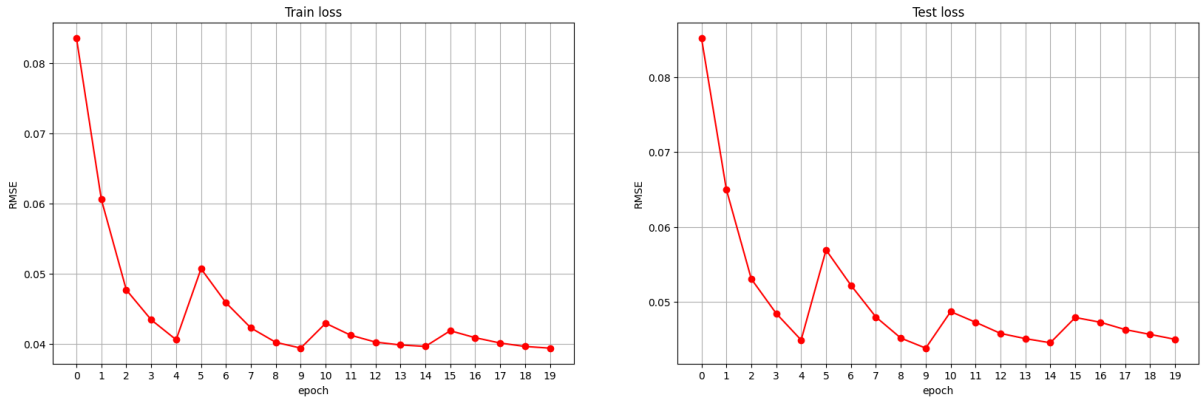


Figure 13: RMSE for KAN [2,2,1] k = 3, g = 10

As appreciably, results of grid refining for this dataset are not as good as for the authors dataset. Even by using the same optimizer they used.

To check whether this is just for grid size 10, we performed the experiment that authors carried out in *Example 1: Function fitting* [2], which consists in training a KAN with grid size 5, and then performing a *grid extension technique*. This strategy has the objective to refine the precision of B-splines that conform the KAN. It is a transfer learning technique in which a KAN with coarse grained grid is trained, and then, more control points are added to its splines to keep training and being more precise. Figure 9 shows the RMSE result of this strategy.

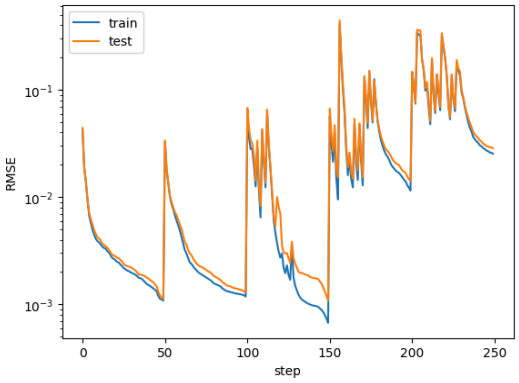


Figure 14: RMSE as function of LBFGS step

In contrast to the authors experiment, we observed that losses don't necessarily fall when refining the grid size for this dataset. For the case of the authors' dataset, they report that for a grid size 1000 their RMSE stopped working, probably due to a bad loss landscape.

Supposedly, the new control points added in each grid refine, are initialized such that they minimize the distance between the coarser function and the new finer function, see figure 2.16 from author's paper, attached also here in Figure 10.

$$\{c_j'\} = \underset{\{c_j'\}}{\operatorname{argmin}} \underset{x\sim p(x)}{\mathbb{E}} \left(\sum_{j=0}^{G_2+k-1} c_j' B_j'(x) - \sum_{i=0}^{G_1+k-1} c_i B_i(x) \right)^2$$

Figure 15: Newly added control points initialization

However, we experienced sudden increases in loss values for every grid extension we performed.

To give a proper origin of the problem, we should investigate much more, however, our main theories are the following:

1) Maybe control points are not being initialized well in the author's implementation, in fact, there is an open issue that states that coefficients (points) may have been initialized twice. Authors have also mentioned in a response to another issue, that there may be a potential bug in the way the grid is updated.
2) It is also possible that this is not the best way to implement coefficient initialization.

Different optimizer

Authors have only tried the LBFGS optimizer during their tests. However, they have also implemented the possibility of using Adam as optimizer. Therefore, we have played with Adam as optimizer to try to minimize the test loss and training time. Our objective was to overpass authors trials with LBFGS by choosing a proper Adam optimizer.

Which learning rate should be chosen?

When using an Adam optimizer, the first doubt question is to choose a learning rate. Normally, this hyperparameter is set to 0.01, however, this may not be equal for KANs since they use a totally

different approach. Since authors have not tried Adam, they have not reported which learning rate is optimal for it.

We have experimented with different learning rates, ranging from 0.2 to 0.001 for a KAN of grid size 3 trained with 500 steps, and splines order 3. For every learning rate, we performed 3 measurements Figures 11 and 12 show the result of our experiment. Both figures show an interesting property of KANS. Learning rate does not behave equally for KANs than for typical neural networks! . It is clear that for typical values of learning rate, KANs do not minimize their loss. Instead, it seems that a proper learning rate should be located somewhere between 0.05 and 0.017. In this experiment, the minimum loss was 0.00175 and was obtained for a learning rate of 0.11.

Figure 16 learning rates vs MSE

```
Mean test errors for different learning rates:
lr = 0.2: mean test loss = 0.0284553412348032
lr = 0.19: mean test loss = 0.006864449474960566
lr = 0.18: mean test loss = 0.010421637445688248
lr = 0.17: mean test loss = 0.0028132779989391565
lr = 0.16: mean test loss = 0.0018767542205750942
lr = 0.15: mean test loss = 0.004115933086723089
lr = 0.14: mean test loss = 0.028651392087340355
lr = 0.13: mean test loss = 0.01651771180331707
lr = 0.12: mean test loss = 0.002915494842454791
lr = 0.11: mean test loss = 0.0017581613501533866
lr = 0.1: mean test loss = 0.0024379717651754618
lr = 0.09: mean test loss = 0.004484244156628847
lr = 0.08: mean test loss = 0.003694540122523904
lr = 0.07: mean test loss = 0.0023788276594132185
lr = 0.06: mean test loss = 0.006348830182105303
lr = 0.05: mean test loss = 0.007399140391498804
lr = 0.04: mean test loss = 0.013362412340939045
lr = 0.03: mean test loss = 0.008444043807685375
lr = 0.02: mean test loss = 0.01064128428697586
lr = 0.01: mean test loss = 0.01010045688599348
lr = 0.009: mean test loss = 0.008356962352991104
lr = 0.008: mean test loss = 0.01129863876849413
lr = 0.007: mean test loss = 0.009599885903298855
lr = 0.006: mean test loss = 0.010900087654590607
lr = 0.005: mean test loss = 0.012450236827135086
lr = 0.004: mean test loss = 0.014505092054605484
lr = 0.003: mean test loss = 0.017627960070967674
lr = 0.002: mean test loss = 0.02506980486214161
lr = 0.001: mean test loss = 0.051705535501241684

Minimum mean loss 0.0017581613501533866 for lr = 0.11
```

Figure 17: Mean and minimum loss over all lr

During all previous different trainings, we noticed that some results differed among the number of steps performed, therefore another trial with a different number of steps was needed. We repeated the experiment for 244 steps. The selection of this number of steps is explained on page 10 (*Can Adam overpass LBFGS for* $f(x, y) = xy$ *?*). Figures 13 and 14 show the results of this experiment. It is clear that RMSEs follow the same structure as the previous case, and now, the optimal one is 0.12. However, graphically it seems that the real minimum of a smooth function of RMSE depending on lr should be located in 0.17 .

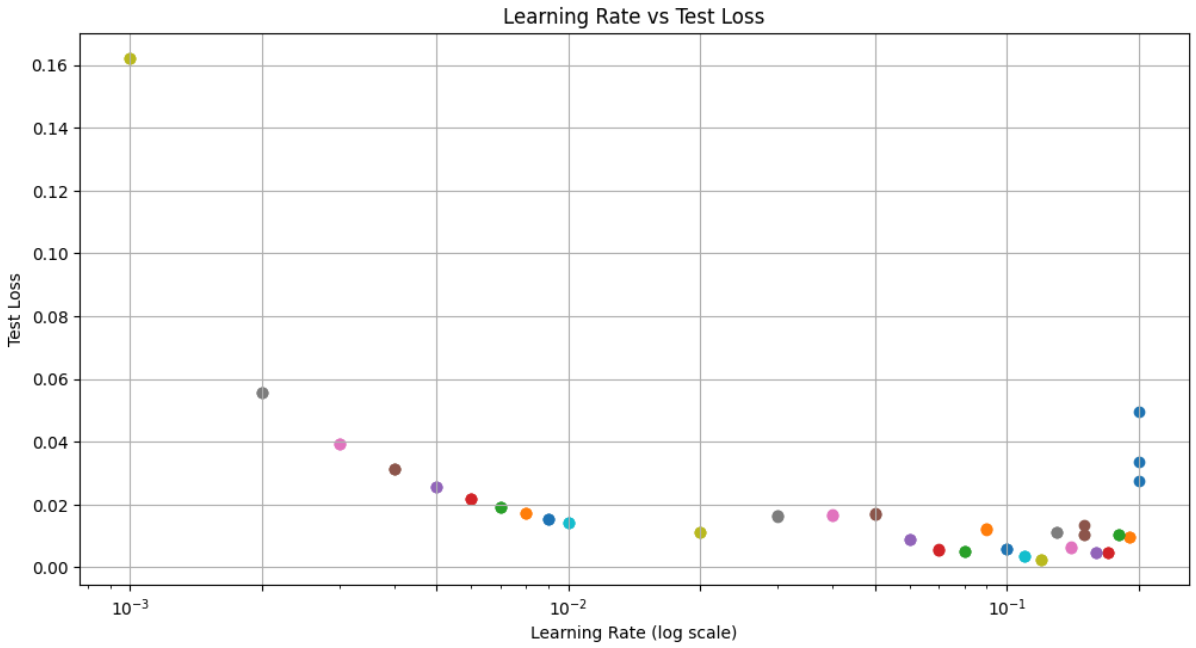


Figure 18: learning rates vs RMSE for 244 steps

```
Mean test errors for different learning rates:
lr = 0.2: mean test loss = 0.0369129478931427
lr = 0.19: mean test loss = 0.009700957685709
lr = 0.18: mean test loss = 0.010363142006099224
lr = 0.17: mean test loss = 0.004562607500702143
lr = 0.16: mean test loss = 0.00453819939866662
lr = 0.15: mean test loss = 0.01141301915049553
lr = 0.14: mean test loss = 0.00640825042501092
lr = 0.13: mean test loss = 0.011205755174160004
lr = 0.12: mean test loss = 0.0023921593092381954
lr = 0.11: mean test loss = 0.0037238358054310083
lr = 0.1: mean test loss = 0.005835516378283501
lr = 0.09: mean test loss = 0.01214937400072813
lr = 0.08: mean test loss = 0.0052636186592280865
lr = 0.07: mean test loss = 0.00554814375936985
lr = 0.06: mean test loss = 0.008895989507436752
lr = 0.05: mean test loss = 0.01708363927900791
lr = 0.04: mean test loss = 0.016710804775357246
lr = 0.03: mean test loss = 0.01644313894212246
lr = 0.02: mean test loss = 0.011283830739557743
lr = 0.01: mean test loss = 0.0140995429828763
lr = 0.009: mean test loss = 0.015485008247196674
lr = 0.008: mean test loss = 0.01727505587041378
lr = 0.007: mean test loss = 0.0190949235111475
lr = 0.006: mean test loss = 0.021721212193369865
lr = 0.005: mean test loss = 0.025707991793751717
lr = 0.004: mean test loss = 0.03143167123198509
lr = 0.003: mean test loss = 0.03932742401957512
lr = 0.002: mean test loss = 0.05565604194998741
lr = 0.001: mean test loss = 0.16207942366600037

Minimum mean loss 0.0023921593092381954 for lr = 0.12
```

Figure 19: Mean and minimum losses for 244 steps of Adam

To finish our learning rate experiments, we decided to experiment with the dataset that authors used for most of their experiments, $f(x, y) = e^{sin(\pi x)+y^2}$. Figures 14 and 15 show the result of this experiment. In this case, we can observe a very similar distribution for RMSEs depending on the learning rate, however, for this function, Adam optimizer seems to work worse than LBFGS. We performed 500 steps of Adam, each training took about double the time of 20 LBFGS steps, but RMSE for Adam was considerably superior to what is shown in the author's experiment for LBFGS. Despite that, in this case, we have obtained that the best learning rate should be 0.18.

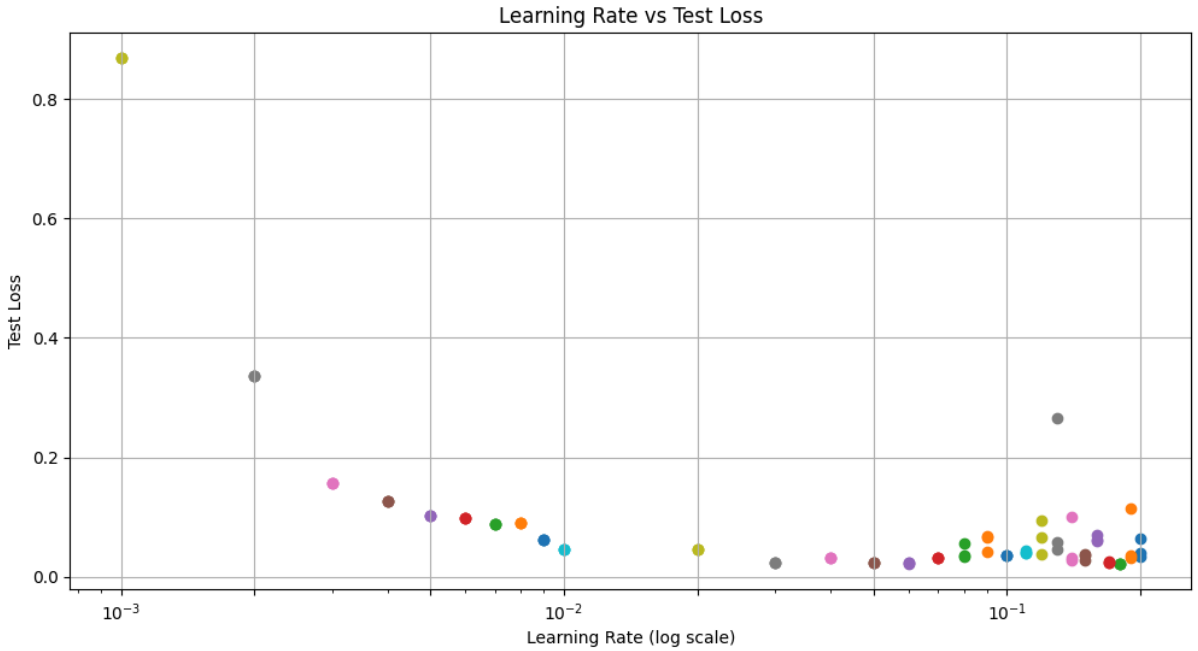


Figure 20 RMSE vs learning rate for 500 Adam steps in $f(x, y) = e^{sin(\pi x)+y^2}$ *dataset*

```
Mean test errors for different learning rates:
lr = 0.2: mean test loss = 0.045901939272880554
lr = 0.19: mean test loss = 0.06004089117050171
lr = 0.18: mean test loss = 0.02185255102813244
lr = 0.17: mean test loss = 0.024895327165722847
lr = 0.16: mean test loss = 0.06367520987987518
lr = 0.15: mean test loss = 0.03442675992846489
lr = 0.14: mean test loss = 0.05370593070983887
lr = 0.13: mean test loss = 0.12322745472192764
lr = 0.12: mean test loss = 0.06687269359827042
lr = 0.11: mean test loss = 0.04069747403264046
lr = 0.1: mean test loss = 0.03525945916771889
lr = 0.09: mean test loss = 0.05858955904841423
lr = 0.08: mean test loss = 0.04198049381375313
lr = 0.07: mean test loss = 0.03240424394607544
lr = 0.06: mean test loss = 0.02282596193253994
lr = 0.05: mean test loss = 0.02385956607758999
lr = 0.04: mean test loss = 0.031025782227516174
lr = 0.03: mean test loss = 0.023893998935818672
lr = 0.02: mean test loss = 0.04542829096317291
lr = 0.01: mean test loss = 0.04535473510622978
lr = 0.009: mean test loss = 0.06253966689109802
lr = 0.008: mean test loss = 0.08947502821683884
lr = 0.007: mean test loss = 0.08845818787813187
lr = 0.006: mean test loss = 0.0986248180270195
lr = 0.005: mean test loss = 0.10290165990591049
lr = 0.004: mean test loss = 0.1264670491218567
lr = 0.003: mean test loss = 0.15627333521842957
lr = 0.002: mean test loss = 0.33664175868034363
lr = 0.001: mean test loss = 0.8683735728263855

Minimum mean loss 0.02185255102813244 for lr = 0.18
```

Figure 21 mean RMSEs for different learning rates

Can Adam overpass LBFGS for $f(x, y) = xy$?

Now that we know that for $f(x, y) = xy$ a proper learning rate should be 0.11 or 0.12 and that LBFGS worked quite well for grid size 3, 20 steps and degree 3.

To perform a fair comparison in terms of computational expenses against LBFGS case, we will consider 244 steps, which correspond to the integer part of 20 * 12.207. Therefore

For this situation, the **training time was 6.09s**, and the last loss obtained was **0.0024, which represents just a fraction of 58% of the first error of 0.00401!!!** . Figure 16 shows this result as well as the learnt network.

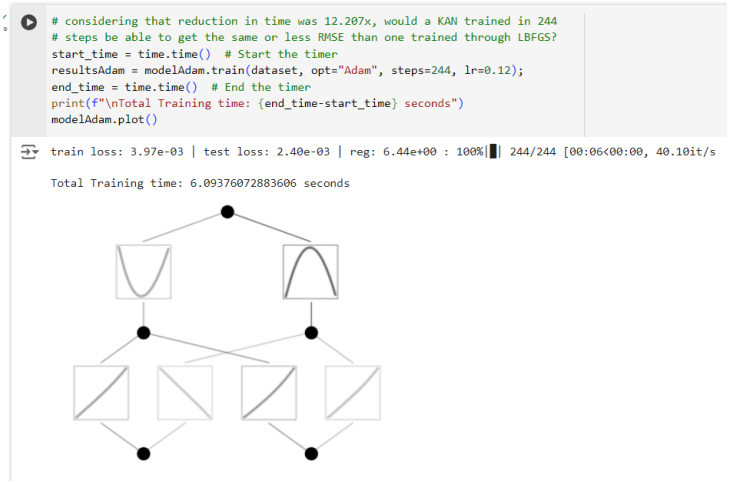


Figure 22: 244 Adam steps for lr = 0.12 in xy dataset

It is impressive how well the learnt architecture represents the real Kolmogorov Arnold decomposition for our dataset $f(x, y) = \frac{1}{4}[(x + y)^2 - (x - y)^2]$. Figure 17 shows the evolution of losses for this experiment.

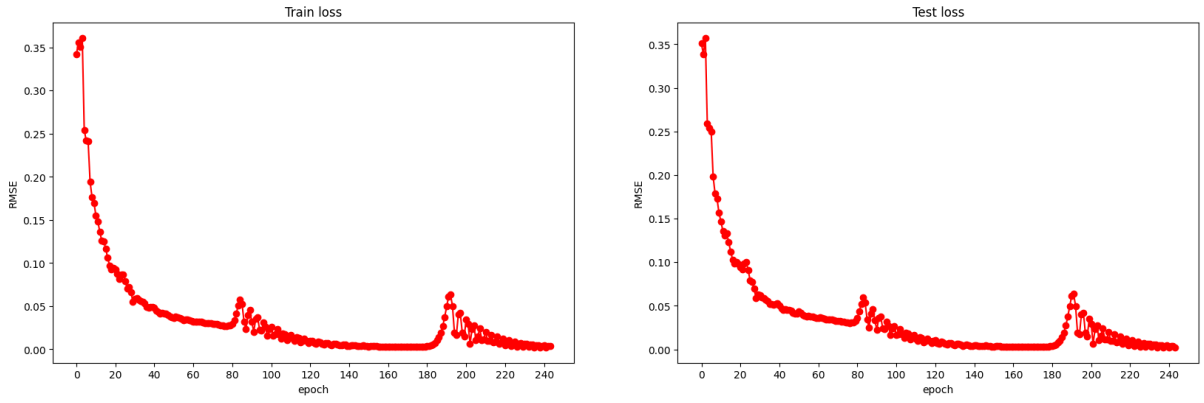


Figure 23: Evolution of losses for 244 Adam steps

To conclude this section, we can state that Adam is able to overpass LBFGS for dataset $f(x, y) = xy$, concretely with a learning rate of 0.12.

How does the grid extension technique behave for Adam optimizer?

We performed the same grid extension technique as shown in the authors' examples. It also did not work quite well. Figure 18 shows the evolution of losses.

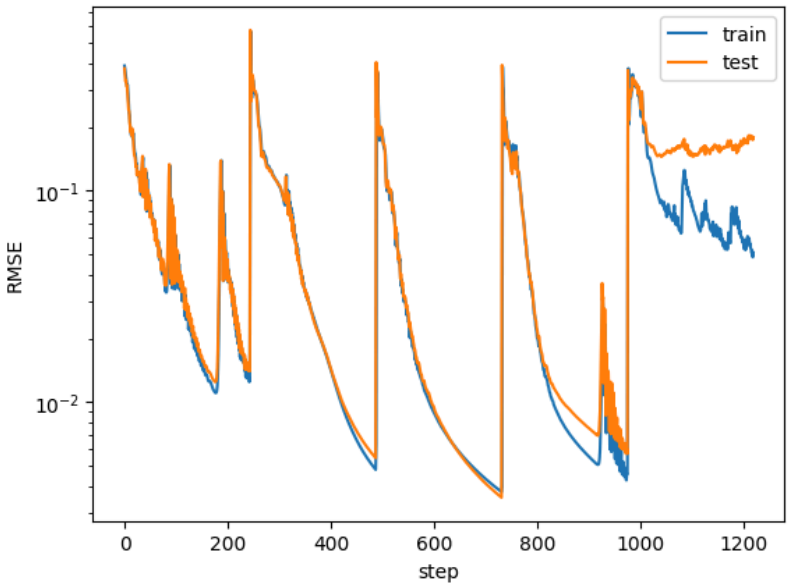


Figure 24: RMSEs evolution for grid technique. Adam is used as optimizer and every grid extension occurs within 244 steps

Despite that, it is clear that a grid extension of 10, works better than a coarser grid extension of 5.

KAN with an unknown function

In the previous example, together with the examples of the paper, the dataset structure was known before the experiment. Having in mind this structure, we chose the best width to fit the dataset. However, our goal is to extend KANs to real datasets where the structure does not follow a clear mathematical function. But before going on with real datasets, we did a last test using a synthetic dataset crafted by us aiming to replicate an unknown scenario in order to see the performance of KANs.

Although the dataset is thought to follow an unknown function, we have used the following function to craft it: $f(x, y) = 2^{cos^2(x)\, sin^2(y)}$.

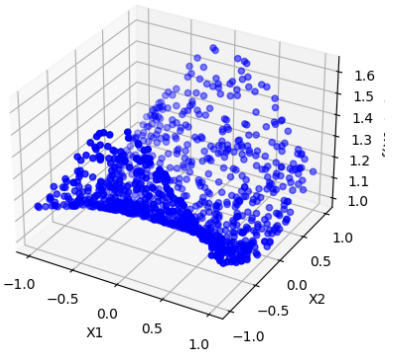


Figure 25: Dataset from an unknown function

We can appreciate in Figure 9 that it has a structure with a saddle point but just from this sample it is difficult to infer the function behind this dataset.

After crafting the dataset, we need to initialize the model, train it and see the performance before fine tuning it. For this model, we have used the following width: [2, 5, 4, 1]. The width was chosen according two statements: firstly, KANs tend to perform better in architectures with only a few layers and secondly, we will try to apply pruning in the best architecture so we need to have such a number of activation functions that it can be considerably reduced in the pruning.

This model has 204 parameters. After training the model, with an execution time of 19 seconds, it has obtained a 7.79e-04 test loss.

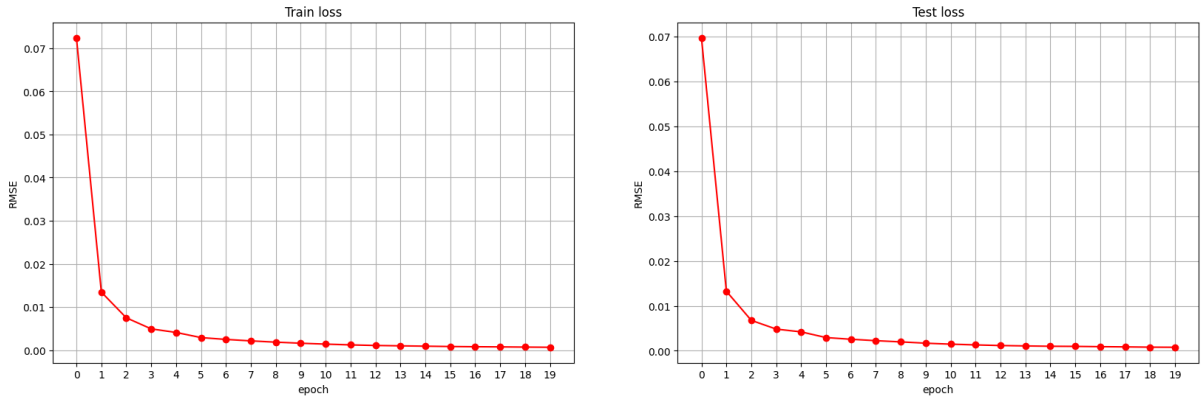


Figure 26: RMSE for KAN and unknown dataset [2,5,4,1] k = 3, g = 3

We can appreciate in Figure 10 that the model has performed really well. With only five epochs, the model managed to achieve a smaller loss than 0.005.

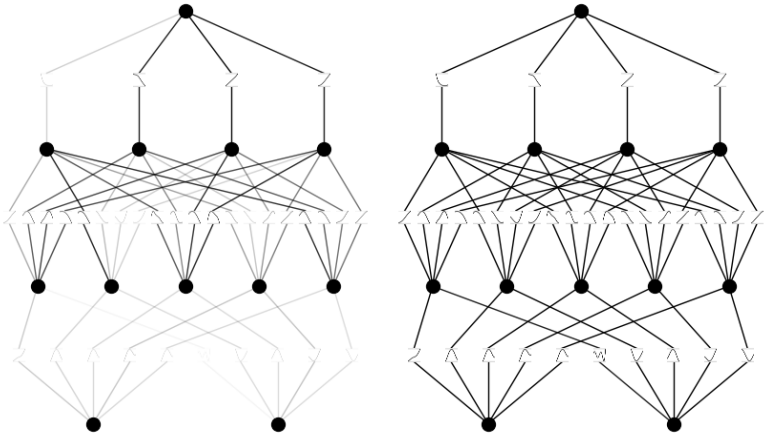


Figure 27: Trained model for an unknown dataset (left) and the same model after refining the grid size in 10 (right)

Refining the grid size

As the paper stated, we can refine the model by retraining it with different grid sizes. If we initialize a new model with the previous trained one, and now we change the grid to 10 and we train it again, the performance of the model has increased. Now, the second model has managed to achieve a test loss of 1.26e-04 in 24 seconds of execution time. Note that the execution time has increased as well as the number of parameters which has been raised to 442 just by changing the grid size from 3 to 10.

We have kept refining the grid with 25, 50, 75 and 100 as grid sizes. We have started by loading the last model which was already refined with 10 as grid size. Then, we have iterated over the grid sizes, initializing the current model with the model trained with the previous grid size. Opposite to the results obtained by the authors with their function, in our case the test loss was better after refining the grid size the first time, that is, after retraining the first model, which was trained with 3 as grid size, with grid size 10. However, after retraining 4 more times the model with the grid sizes stated above, the test loss has been worse (Table 1 and Figure 12). So, this experiment has led us to the same conclusion as the previous experiment which was that refining the model with more grid sizes is not always a good choice for improving the performance of the model.

Iteration	Refined Grid Size	Test Loss	Number of parameters	Execution Time (s)
1	25	1.31e-03	952	88.01
2	50	4.70e-03	1802	137.42
3	75	2.77e-02	2652	193.82
4	100	1.03e-02	3502	232.14

Table 1: Concatenated refined grid sizes

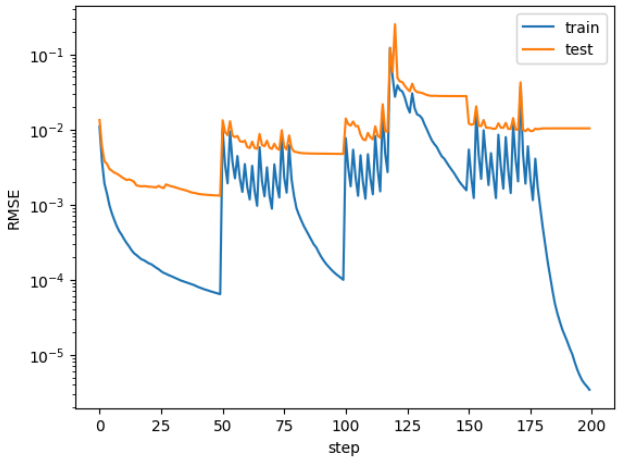


Figure 28: Evolution of losses during the different grid refinings

Different KAN Architectures

The next step in order to improve our model is to try different architectures to analyze the differences between their performances.

We tried the following architectures, all with LBFGS optimizer, and we can see the results in Table x.

ID	Width	#params	#steps	Grid size	Test loss	Execution time (s)
0	2,4,4,1	364	100	10	2.04e-04	112.27
1	2, 10, 5, 1	975	100	10	1.79e-04	250.35
2	2, 2, 1	78	100	10	1.90e-03	37.94
3	2, 7, 1	273	100	10	1.73e-04	72.66

Table 2: Different architecture executions with LBFGS

As we can see, wider layers at the beginning performed better in terms of test loss compared to narrower initial layers (model id 2), as wider layers can capture more complex features early in the network. Models that transitioned to narrower layers towards the end also showed good performance, so broad feature extraction followed by specific fine-tuning could be an effective strategy.

There is a clear trade-off between the number of parameters and execution time. We can see how models with higher numbers of parameters performed better but had longer execution times, so the balance between accuracy and complexity is quite important.

We can see the performance of one of these models for reference in the following graph, specifically of the third model.

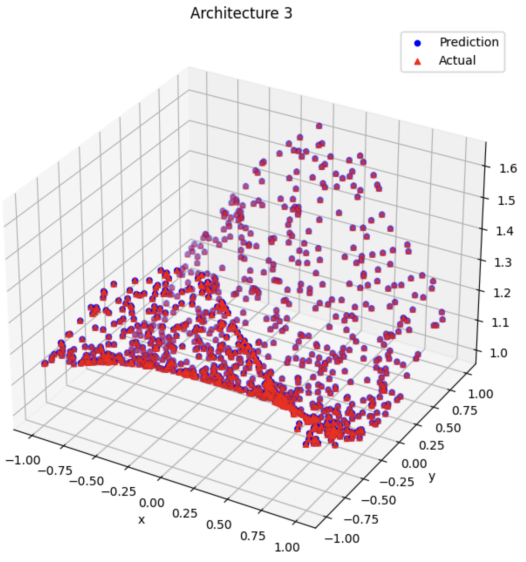


Figure 29: Actual and Predicted points

Pruning different KAN Architectures

After trying different architectures, we wanted to explore the interpretability of KANs through one of the main features of this network: the pruning.

Pruning is used for simplifying the KAN by deleting neurons or connections considered less important given a certain threshold. This way, we can reduce the complexity of the model without significantly sacrificing its performance.

The idea behind this is improving efficiency, because if the number of neurons and connections is reduced, the network becomes less computationally expensive, which can speed up the training process, and reducing the overfitting, because we would be deleting nodes that may be capturing noise rather than meaningful patterns in the data.

After pruning these models, we obtained the following results.

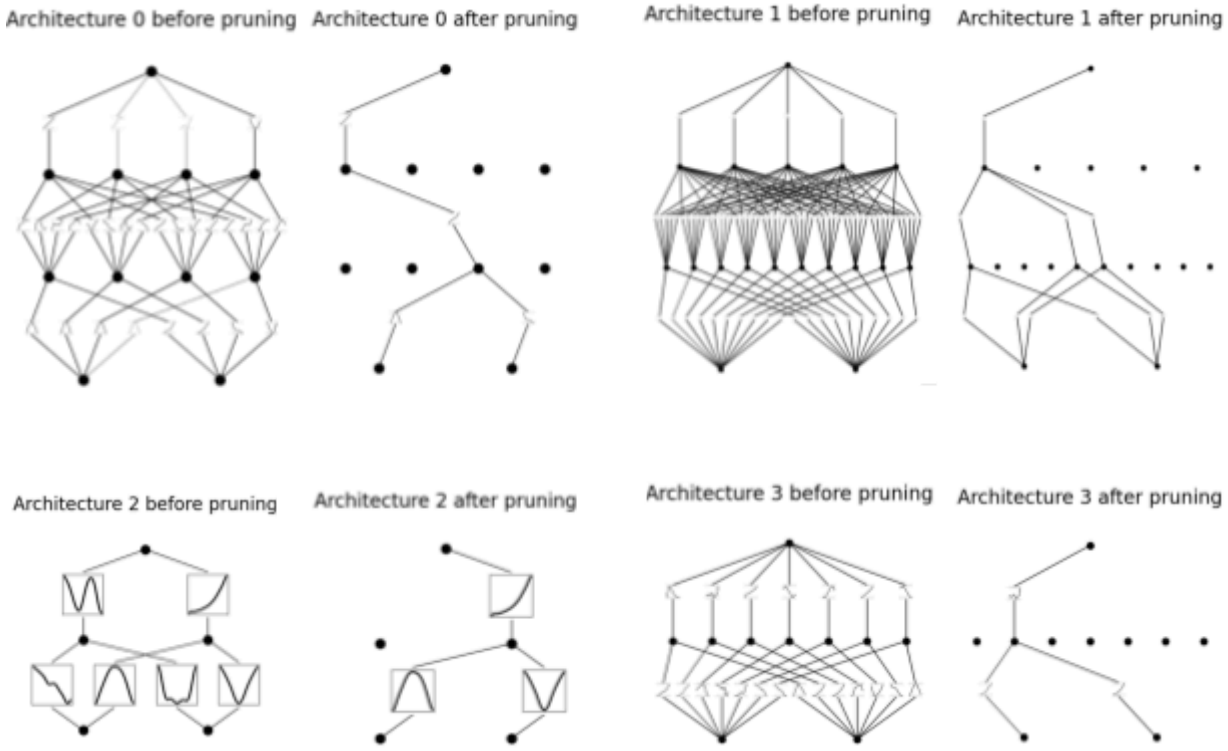


Figure 30: Different architectures before and after pruning

As we can see, pruning different KAN architectures leads to a significant reduction in model complexity by decreasing the number of neurons and connections. It is interesting to observe that different initial architectures can lead to the similar final architecture, as it happens with models 0 and 1 and with models 2 and 3, which have the exact same architecture and also have similar activation functions. This is a good sign, as it indicates that the pruning process effectively identifies and retains the most critical components of the network, regardless of the starting structure. This convergence to a similar pruned architecture suggests that the KAN is capturing effectively the robust underlying structure within the data.

However, we can see in the following image the performance of the predictions after pruning of the third model, for example.

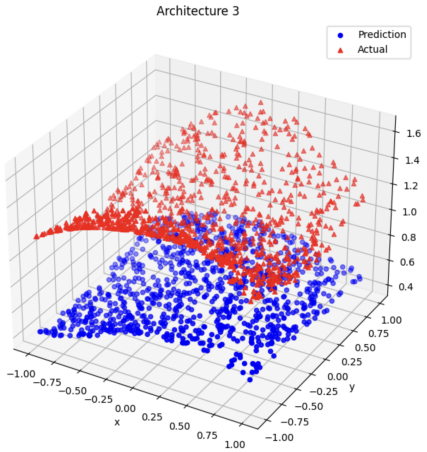


Figure 31: Performance after the pruning of the third model

As we can see, the performance of the model after pruning is quite bad, there is a huge difference between the predicted and the actual values.

So we decided to train again the pruned model and we obtained the following results, with a test loss of 7.15e-02, 100 higher than the previous one, 1.73e-04.

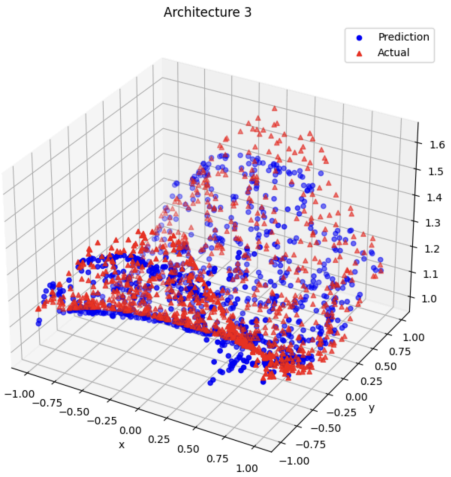


Figure 32: Performance after the pruning of the third model retrained

Considering all this, we can say that pruning simplifies model complexity and makes it more interpretable, but it significantly degrades accuracy, making pruned models less useful for prediction tasks. Something interesting that we saw during these trials it's that when all of the neurons of a certain layer get deleted, the model loses the ability of predicting the output, so this also indicates that pruning is not really meant for using the model afterwards, just for improving the interpretability. However, that can be controlled by the threshold parameter and additionally, future work might focus on finding a balance between pruning and performance, potentially incorporating techniques such as regularization or selective retraining to mitigate the negative impacts on accuracy.

Comparing KAN vs MLP

In the paper, the authors stated that KANs overpass MLPs in some scenarios with results that are nearly 100 times more accurate and 100 times more parameter efficient. However, they studied this comparison using LBFGS as optimizer, leaving as "Future Work" the comparison between the performance of KAN using Adam as optimizer and the traditional MLPs.

To achieve this goal, we tried different architectures as well as different parameters of KAN models with Adam to find the best combinations for our experiment. We chose the parameters according to the results of our previous work, that is, we chose 0.12 as the learning rate and 10 as grid size. Results from several executions can be visualized in Table 2.

ID	Width	#params	#steps	Grid size	Test loss	Execution time (s)
0	2,4,2,1	234	20	10	1.06e-03	15.14
1	2,4,2,1	234	50	10	3.38e-04	36.88
2	**2,4,2,1**	**234**	**100**	**10**	**2.20e-04**	**73.89**
3	2,4,2,1	234	200	10	1.66e-04	128.79
4	2, 1, 2, 1	78	20	10	9.23e-02	8.98
5	2, 1, 2, 1	78	50	10	9.49e-02	21.67
6	2, 1, 2, 1	78	100	10	9.14e-02	43.37
7	2, 1, 2, 1	78	200	10	9.66e-02	67.82
8	2, 6, 1, 2, 3, 1, 3, 1	455	20	10	3.64e-02	32.34
9	2, 6, 1, 2, 3, 1, 3, 1	455	50	10	4.37e-02	81.59
10	2, 6, 1, 2, 3, 1, 3, 1	455	100	10	4.00e-02	159.07
11	2, 6, 1, 2, 3, 1, 3, 1	455	200	10	5.60e-02	318.64
12	2, 7, 1	273	20	10	5.23e-04	14.29
13	2, 7, 1	273	50	10	3.06e-04	33.85
14	**2, 7, 1**	**273**	**100**	**10**	**1.82e-04**	**66.75**
15	2, 7, 1	273	200	10	1.12e-04	132.87
16	2, 5, 5, 5, 1	845	20	10	9.97e-03	45.61
17	2, 5, 5, 5, 1	845	50	10	8.35e-03	116.53
18	2, 5, 5, 5, 1	845	100	10	6.56e-03	225.24
19	2, 5, 5, 5, 1	845	200	10	5.37e-03	437.01

Table 3: Different executions with Adam

In order to compare the architectures with MLP, we created 5 MLP models with a similar number of parameters (equal or higher, but not lower) for each of the top five KAN architectures seen before, without pruning. The shape of the MLP Architecture is *input_size, [hidden_sizes], output_size*.

KAN Architecture	#params	Test loss	Execution time	MLP Architecture	#Params	Test loss	Execution time
2, 4, 2, 1	234	2.20e-04	129.54	2, [16, 16, 8], 1	465	0.0936	2.26
2, 1, 2, 1	78	9.33e-02	91.60	2, [12, 4], 1	107	0.0243	1.80
2, 6, 1, 2, 3, 1, 3, 1	455	4.53e-02	229.64	2, [20, 24], 1	589	0.0137	1.91
2, 7, 1	273	1.74e-04	95.00	2, [16, 16], 1	337	0.0124	1.78
2, 5, 5, 5, 1	845	5.73e-03	308.32	2, [32, 32], 1	1185	0.1188	2.74

Table 4: Comparison between KANs and MLPs

After analyzing the test losses and the execution time, we have reached several conclusions. First, the majority of KAN models can achieve about 100 times lower test losses compared to their corresponding MLP models. This suggests that KAN models may be more effective in capturing the underlying patterns in the data.

Secondly, MLP models train significantly faster than KAN models. This is no surprise since it was the thesis of KANs from the very beginning.

Another interesting fact is that all MLP models have more parameters than the corresponding KAN models, and still the KAN models provide a better performance (lower test loss). Also, increasing the number of training steps generally improves performance initially, but there is a risk of overfitting as steps increase beyond a certain point.

We can also infer from Table 3 how placing wider layers at the beginning improves the model's performance, as it can capture more complex features earlier. This conclusion will be reflected multiple times in the other experiments conducted.

Summarizing, if the primary goal is to achieve the lowest test loss, KAN models are more effective despite their longer training times. However, if training time is a critical factor, MLP models offer a substantial speed advantage.

KANs on real datasets

As stated before, one of the key goals of this work is to check the performance of KANs on real datasets. When we began the project, to the best of our knowledge it was only tested with Iris dataset[8] on classification tasks. We have tried to perform classification tasks on other real datasets. First, we used the Titanic dataset to see how KANs works with binary classification. However, we could not achieve great accuracy. Probably, due to the fact that the vast majority of the fields are non-numerical and working with one-hot encodings may not be a great solution for working with KANs. However, we should keep investigating whether or not our hypothesis is the reason for this failure case.

We opted for changing the dataset to another one with numerical fields. We used a heart disease dataset in order to predict whether or not a patient has a cardiac disease, again with a binary classification problem.

Binary Classification: Heart disease

This heart disease dataset[9] is created by merging five well-known heart disease datasets that were previously available separately but had not been combined before (Cleveland, Hungarian, Switzerland, Long Beach VA and Statlog (Heart) Dataset).

The dataset comprises 1190 instances with 11 features. The features are the following: age, sex, chest pain type, resting bps, cholesterol, fasting blood sugar, resting ecg, max heart rate, exercise angina, oldpeak and ST slope. The label of this dataset is called 'target' and it can be 0 or 1.

Having in mind all the parameter study done before in this project, we build several KAN architectures with different parameters to test the results for this task. The best architectures we found had a width of [11,3,1], [11,15,3,1] and [11,15,5,1] (can be seen in Table 4). Over all the best executions, there is one that outperformed. This one has width [11,15,5,1], 10 as grid size, grade 4 for the splines, 100 steps and 0.05 as learning rate. With this configuration we achieved 100% of accuracy with the train set and a test accuracy over 93%. Although it is obvious that there is overfitting in this case, the model has amazing results for the testing dataset.

ID	Width	Grid size	Spline grade	#Steps	Learning rate	Train accuracy	Test accuracy	Execution time (s)
0	[11,3,1]	10	4	100	0.12	0.8771	0.8949	12.35
1	[11,3,1]	10	3	200	0.12	0.8771	0.8739	20.25
2	[11,3,1]	10	4	200	0.05	0.8708	0.8866	21.40
3	[11, 15, 3, 1]	10	3	200	0.12	0.96	0.8782	90.89
4	[11, 15, 3, 1]	10	4	200	0.12	0.9286	0.8824	102.86
5	**[11, 15, 5, 1]**	**10**	**4**	**100**	**0.05**	**1.0**	**0.9328**	**99.76**
6	[11, 15, 5, 1]	10	3	200	0.12	0.9370	0.8739	53.14
7	[11, 15, 5, 1]	20	4	200	0.05	0.9958	0.8655	178.25

Table 5: Best executions with the Heart Disease dataset

Using the best model mentioned above, we pruned the model to visualize which neurons have the largest weight on the final output. The result can be seen in Figure 13. Note that the model was pruned with the default threshold parameter which is 0.01. We can see how from the second layer to the last one, there is only one activation function per layer that is responsible for all the weight of its layer, drastically reducing the complexity of the model. With this medical example, the interpretability of KANs plays a significant role in failure scenarios. If the network is producing incorrect outputs, it is easy to look at the model and retrieve which neurons have more weight than others in order to find

where the error is lying, and consequently, to fix it. On conventional network architectures, their complexity and lack of interpretability makes it much more difficult the task of finding an error in the whole network.

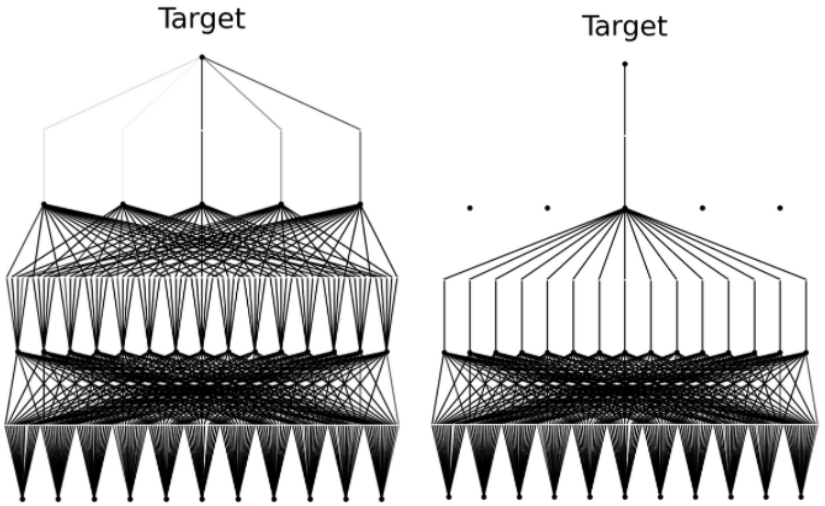


Figure 33: On the left, the model [11,15,5,1] without pruning and on the right, the same model after pruning it

CNN with KAN

Could KAN be used for computer vision tasks? Well at first glance, one could think that KANS are too slow to be able to perform well for CV tasks. Previous judgment will be tested within this section.

Task

Image classification. We decided to use KANs for a simple image classification problem.

Datasets

To perform this task, we will be using the well known *SVHN* and *MNIST* datasets with the data provided for *Lab 3* during this Deep *Learning course*.

How was it achieved?

The first problem we need to solve is the slow implementation of authors, which would have made it almost impossible to perform this. To solve it, seeing that almost a month had passed from paper publication, we decided to search for a more efficient implementation of KANs. We found the *efficient-KAN* repository [5]. Which promises to achieve a way better performance than the original implementation. That repository is also much smaller and interpretable, making it possible for us to adapt to it fast.

Architecture

Our approach here consisted in substituting the last fully connected layers of a traditional CNN with a KAN layer, as seen in figures 13 and 14.

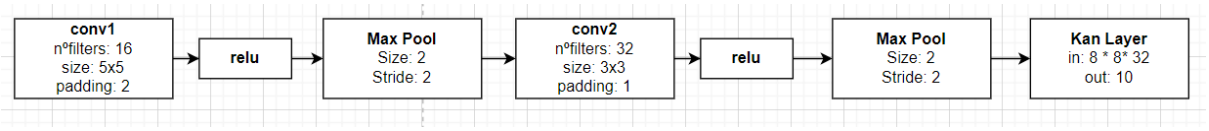


Figure 13: Architecture used for the SVHN dataset

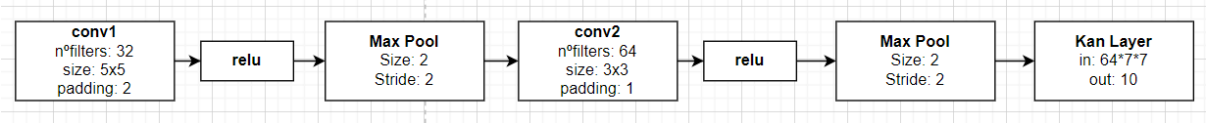
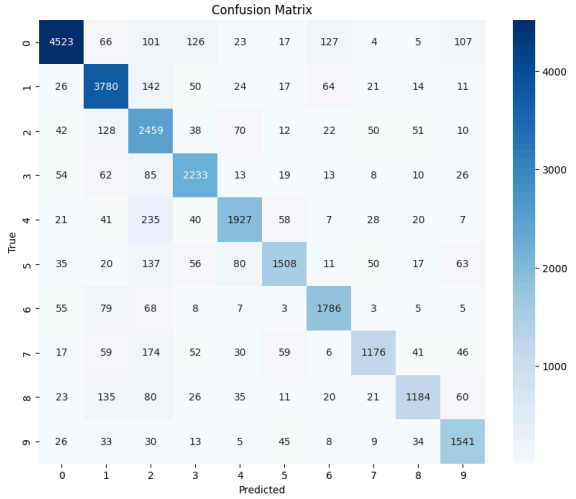


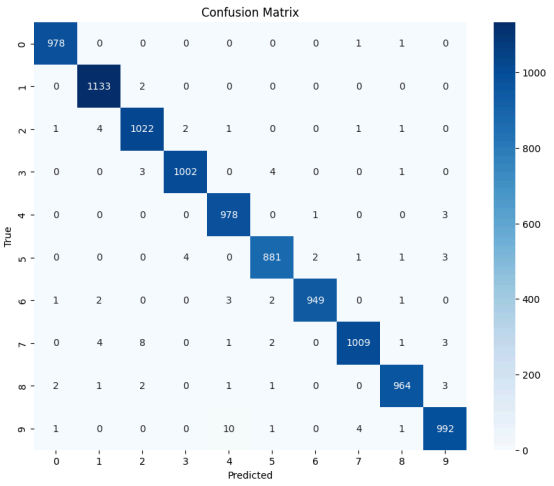
Figure 34: Architecture used for the MNIST dataset

Our KAN layers used a **grid of size 5, and splines of order 3**, therefore having **8 control points** each one.
Therefore, we had **210656 parameters** for the **SVHN** architecture and **332928 for MNIST** architecture. That may sound like a lot of parameters! As stated in the introduction, for every edge of the last layer, we have to add 8 control points in this case.

Performance

Figures 15 and 16 show the confusion matrix, training times and accuracies obtained for both datasets. As one may think due to the truly high number of parameters, training time was not that extensive. Training times were quite high, but later on this section we will see that they were not much higher than for a classical CNN approach. Accuracy was also impressive, especially considering the low complexity of the network.






```
Training time was: 149.82080793380737
Accuracy KAN_CNN_SVHN: 84.96081745543945
```

```
Training time was: 108.87103819847107
Accuracy KAN CNN MNIST: 99.08
```

Figure 35 (left): CNN+KAN confusion matrix, training time and accuracy for SVHN.
Figure 36 (right): CNN+KAN confusion matrix, training time and accuracy for MNIS

It is important to comment that both cases were trained with SGD optimizer since Adam performed much worse and LBFGS is not directly implemented for Efficient KAN. We used an average learning rate of 0.1. It would remain as a future work to test different architectures and different optimizers with different learning rates.

Comparison with a classical CNN

To compare the previous approach with a classical CNN, we used practically the same architecture except for the last layer, which was a fully connected layer. Figures 17 and 18 show both architectures.

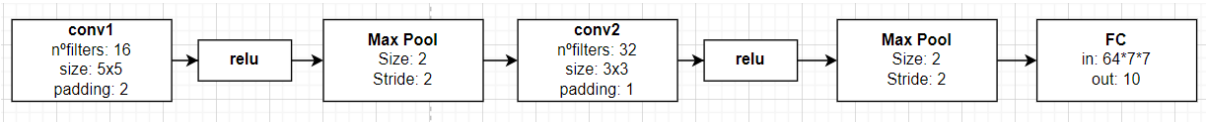


Figure 37: Classical CNN architecture used for SVHN

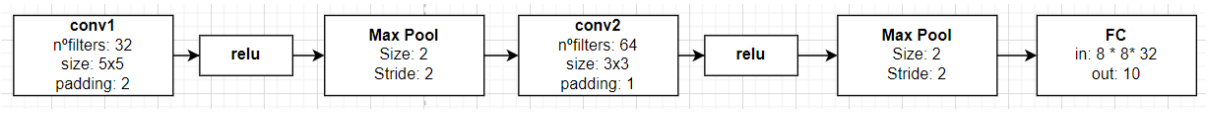
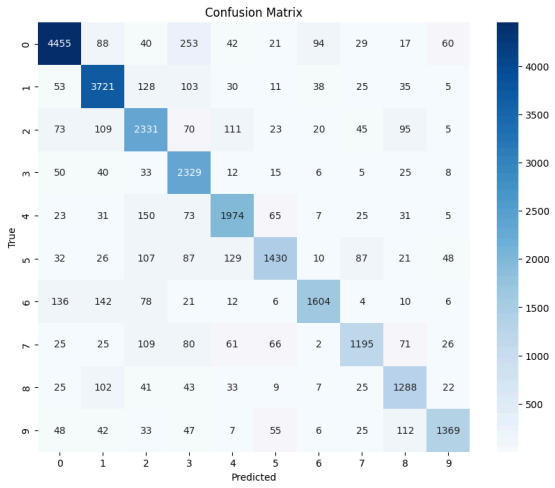


Figure 38: Classical CNN architecture used for MNIST

In this case, **SVHN** architecture is using **26346 parameters** and **MNIST** architecture **20746 parameters**, which as expected, corresponds to 8 times less than previous approach, since no control points are needed.

Figures 19 and 20 show training results. We can clearly appreciate that in both cases, training time is lower than for KANs, which is normal considering the vast amount of extra parameters that KAN layers need to learn. However, it is interesting to notice that the extra amount of parameters doesn't linearly correlate with the extra time needed, in fact **relative to the number of parameters, the KAN approach is much faster!!**. We can also appreciate that accuracy in both cases of traditional FC approach is lower than for KANs approach.



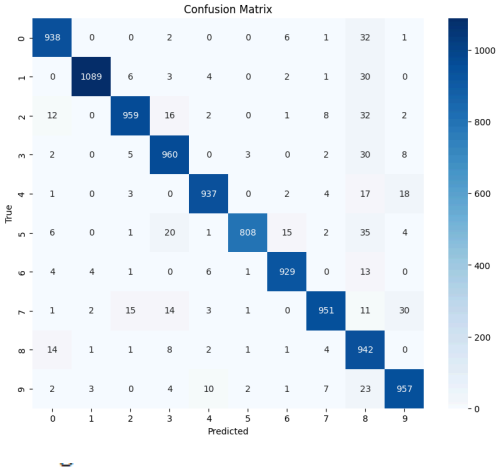


Figure 39 (left): CNN confusion matrix, training time and accuracy for SVHN.
Figure 40 (right): CNN confusion matrix, training time and accuracy for MNIST

Table of comparisons

	# Parameters	**Training time**	**Test Accuracy**
CNN+KAN (SVHN)	210656	149,82	84,96
CNN+FC (SVHN)	26346	117,452	83,34
CNN + KAN (MNIST)	332928	108,87	99,08
CNN + FC (MNIST)	20746	70,84	94,7

Table 6: CNN+KAN vs CNN+FC tests for SVHN and MNIST

Conclusions

Results obtained indicate that KAN layers can be utilized within classical CNN architectures, and even, get some good results. It is early to state that KAN layers are an improvement of FC layers for computer vision tasks.
Nevertheless, KANs have been released very recently, which may lead to some problems in the actual implementation. In fact, at the end of the project, we encountered a novel implementation, *Fast KAN* [6] which promises to be 3,33x faster than *Efficient KAN* [5] used in our implementation. For that reason and seeing the results we obtained for this section, we believe that KANs have potential for substituting FC layers in CNNs.

We leave as a future work to test this approach with *Fast KAN* [6] implementation.

Our contributions to previous work

Contributions to the paper

During the realization of this project, we have made some contributions to the author's work. The main ones are:

1) Establishing a proper learning rate range for Adam optimizer regarding different kinds of problems.
2) Conclude that Adam tends to perform better than LBFGS in most of the datasets.
3) Comparing KANs vs MLPs with Adam optimizer rather than LBFGS
4) Shown that diamond shaped configuration of KANs (intermediate input, wide first layers and thin last layers) tend to be the best ones.
5) Demonstrate that KAN layers can be used to substitute Fully Connected layers in CNNs. However, this still needs more investigation.

Contributions to its implementation

During the process of our work, we have found several limitations of the author's library 'pykan'. These limitations are mostly trivial and are not related with the implementation of KAN architectures which is perfectly correct. However, we believe that improving these small limitations will push to another level the functionality of this python library.

Firstly, the authors always perform the square root to the loss values, independently of which type of loss function we are using. In fact, the authors in their experiments always use the RMSE to express the different loss values. However, even though the implementation of the library can accept a great variety of loss functions, the output of the loss function, despite the function the user chose, is always returned after applying the square root to it. We think that including the option to the user to choose whether or not to apply the square root to the loss value will provide greater flexibility in fine-tuning the model, allowing for adjustments based on specific requirements or preferences for how the loss is interpreted and managed during the training process.

On the other hand, we do believe that there may be a problem in the initialization of newly added grid points when performing a grid extension, since errors shoot up instead of starting from the same point. Thus, another way of initializing the points is more convenient. We will leave this part as future work.

Finally, the visualization of the architectures should be improved. If you build a simple KAN architecture with a few layers and a small number of activation functions, the different plots of the whole network are good looking. Despite this, if your network is more complex (but not a lot more complex) it is impossible to read the different field names of the input layer or the shape of the different activation functions. Although there is an option to save the shape of the model and all the figures of the activation function after each iteration, they represent plenty of different pictures and it is not feasible for the user to verify manually one by one to visualize the final output, for example.

In summary, we have detected limitations such as the returned square root for the loss value or the visualization tools that are, in terms of functionality and correctness of the implementation, trivial. However, it may be worth trying to spend some time trying to find another way of initializing the control points when doing a grid extension. Covering these limitations will significantly enhance the implementation of this library, making it more effective in the study of KANs.

Future work

Since KANs are an extremely new investigation topic in the Deep Learning community, they still have a lot of work to be made. These are some of our suggestions.

1) Work with different residual activation functions and try different optimization tools to get similar or even better results.
2) Compare KANs vs MLPs in different architectures and with different optimizers.
3) Try to improve the initialization of control points when performing the *grid extension* technique.
4) Compare CNN+KAN and CNN+FC approaches with different architectures and different optimizers. Try to maximize the benefits of KANs for computer vision tasks.
5) Implement KANs for RNN models.
6) Test KAN networks performance in quantum computational contexts, since they need a lot of computational speed, but not that much memory as traditional neural networks.

Final Remarks

Our investigation into Kolmogorov-Arnold Networks (KANs) has yielded some insights in the understanding and application of these novel neural networks. Throughout this report, we have studied whether KANs have potential or not to surpass traditional multilayer perceptrons (MLPs), as well as provided some relevant information on how to use KAN networks properly. Our exploration covered diverse areas such as function fitting, binary classification, and the integration of KANs with convolutional neural networks (CNNs).

As a brief summary of our work, we established an optimal learning rate range for the Adam optimizer specific to KANs, highlighted the superior performance of Adam over LBFGS across multiple datasets, and showcased the effectiveness of a diamond-shaped KAN configuration. We also performed some pruning experiments to explore the interpretability of KAN models. Additionally, we concluded that KANs tend to be more accurate than MLPs with less parameters, however, its training time is considerably superior. We have performed these tests with Adam optimizer since the authors did them with LBFGS. Finally, we demonstrated that KAN layers could successfully replace fully connected layers in CNN architectures, although further research is required to fully optimize them in this context.

By checking these results, we can conclude that KAN networks have potential to surpass traditional MLPs, however, more research needs to be done. In fact, KANs may present a potential use case in quantum computing. Given their architecture, KANs are not memory exhaustive, which makes them suitable for quantum environments where memory resources can be limited. However, they do require substantial computational speed to function effectively, aligning well with the high computational capabilities of quantum systems.

Looking forward, we propose several avenues for future work. These include experimenting with different residual activation functions, optimizing control point initialization for grid extension, and extending the application of KANs to recurrent neural networks (RNNs) and quantum computing contexts. Additionally, further comparisons between KANs and MLPs across varied architectures and optimizers will be essential in solidifying the advantages of KANs.

References

[1] Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., Hou, T. Y., & Tegmark, M. (2024). Kolmogorov–Arnold Networks 1. Preprint arXiv:2404.19756v3. From https://arxiv.org/abs/2404.19756

[2] Kind Xiaoming. *Examples*. Retrieved June 11, 2024, from https://kindxiaoming.github.io/pykan/examples.html#

[3] Kind Xiaoming. *Example 1: Function fitting*. Retrieved June 11, 2024, from https://kindxiaoming.github.io/pykan/Examples/Example_1_function_fitting.html

[4] Wikipedia contributors. *L-BFGS*. In Wikipedia, The Free Encyclopedia. Retrieved June 11, 2024, from https://es.wikipedia.org/wiki/L-BFGS

[5] Blealtan. *efficient-kan*. GitHub. Retrieved June 11, 2024, from https://github.com/Blealtan/efficient-kan

[6] Ziyao Li. *fast-kan*. GitHub. Retrieved June 11, 2024, from https://github.com/ZiyaoLi/fast-kan

[7] BSplines.org. *Flavors and types of B-splines*. Retrieved June 11, 2024, from https://bsplines.org/flavors-and-types-of-b-splines/

[8] Team Daniel. *KAN classification*. GitHub. Retrieved June 11, 2024, from https://github.com/team-daniel/KAN/blob/master/KAN_classification.ipynb

[9] Mexwell. *Heart disease dataset*. Kaggle. Retrieved June 11, 2024, from https://www.kaggle.com/datasets/mexwell/heart-disease-dataset