Analysis of an article about KAN - a fundamentally new neural network architecture

05/02/2024

At the heart of all deep learning architectures is a multilayer perceptron (MLP). It has weights and neurons in which activation functions are located. Scientists have been using this paradigm since 1957, when it was proposed by Frank Rosenblatt.

Now, 67 years later, MIT researchers **have presented** an alternative to MLP - a new neural network architecture, called Kolmogorov-Arnold Networks (KAN), which implements the movement of activations to the "edges" of the network.

Not even a day has passed since the article was published, but it has already turned into a sensation. And it's not surprising, because changes in the perceptron paradigm entail changes throughout deep learning, and can turn over large language models and computer vision systems.

However, if you want to understand in more detail how the new product works, and you look into the article, you will see there are almost 50 pages of text, including many complex formulas and notations.

In general, this article will help you understand the KAN device and not go crazy. Go!

How does a perceptron work?

First, let's remember the basic thing: neural networks work with functions. In any supervised learning problem, the neural network has a training sample consisting of pairs $\{x_i, y_i\}$, where x is the input data and y is the "response". The task of the network is to find a multidimensional function f such that $f(x_i) \approx y_i$ for all points in space. In other words, the neural network tries to find a function that summarizes the relationship between the inputs and outputs of the task.

The architecture of a classical perceptron involves searching for such a function using linear layers on which multiplication of inputs by edge weights is performed, and activation functions in neurons.

This architecture is based on Tsybenko's theorem (universal approximation theorem), which proves that a neural network can approximate any continuous function with any accuracy. However, there are other theorems related to function approximation. One of them, the Kolmogorov-Arnold theorem, is precisely what KAN is related to.

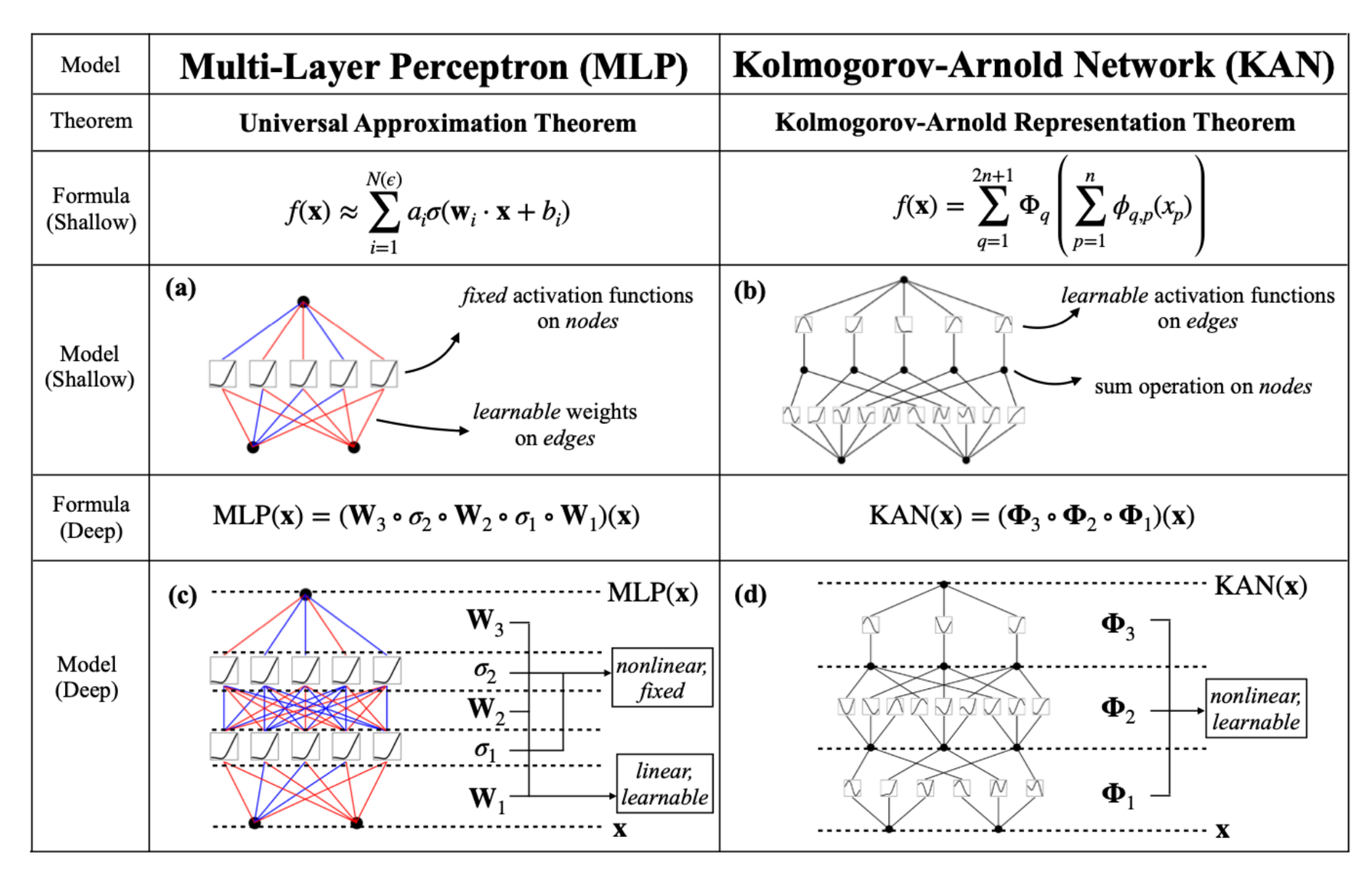


Figure 0.1: Multi-Layer Perceptrons (MLPs) vs. Kolmogorov-Arnold Networks (KANs)

Kolmogorov-Arnold theorem

To accurately understand the structure of KAN, you need to understand mathematics. But we promise, this part will not be boring and not at all difficult.

So, the merit of Kolmogorov and Arnold is that they proved that the approximation of a continuous bounded function of a set of variables reduces to finding the polynomial number of one-dimensional functions:

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

It would seem: this is great news for machine learning: it turns out that in order to "recreate" the big scary function of connection between the inputs and outputs of the network, we need ordinary one-dimensional functions, the number of which also grows polynomially and not exponentially with increasing parameters.

However, not all so simple.

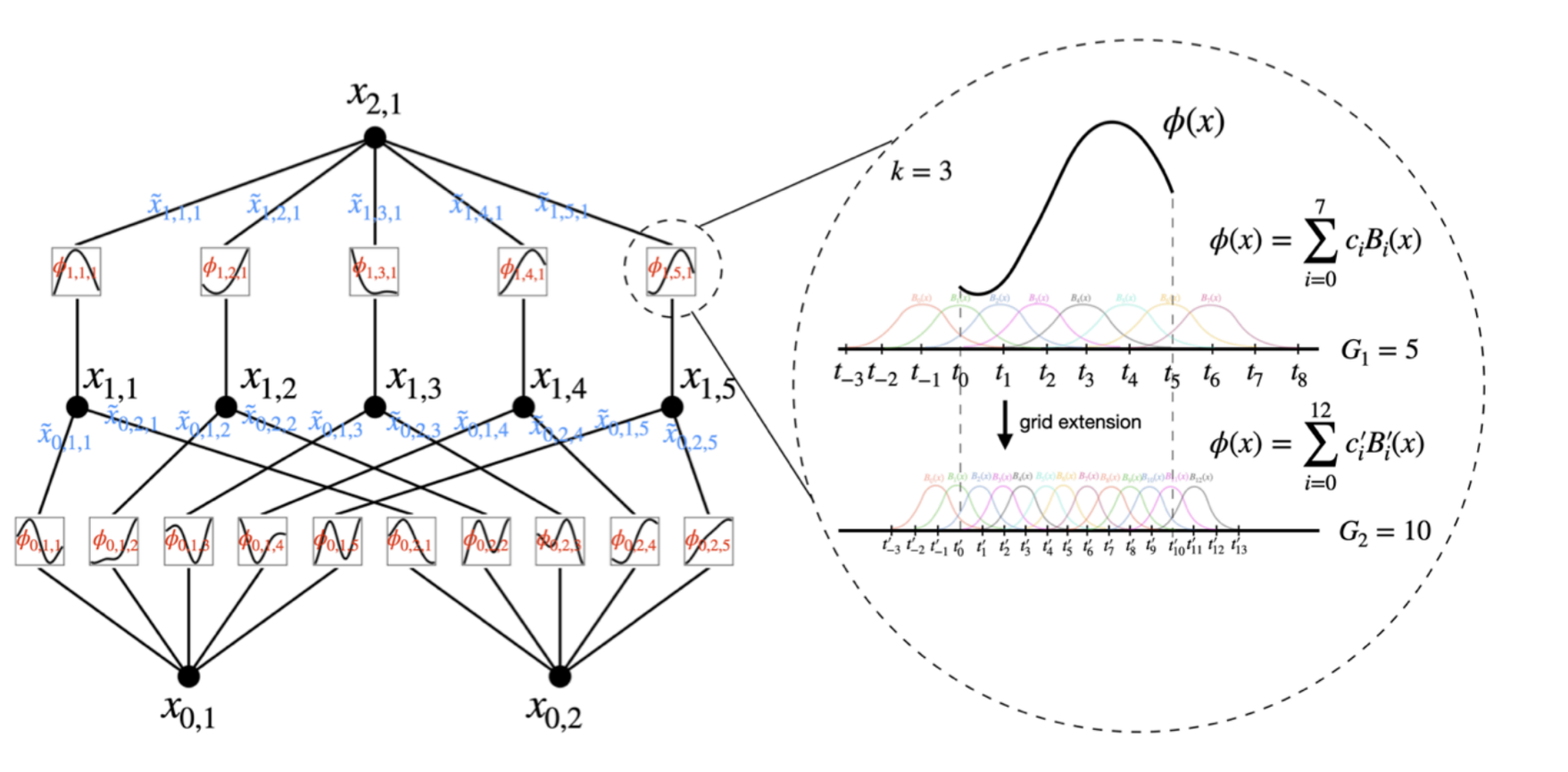
- First, our one-dimensional functions may turn out to be non-smooth and even fractal, and they will be impossible to train.
- Secondly, the number of such functions and the depth of composition in the theorem are fixed, which means our neural network will always have the same size (2 layers and 2n+1 neurons on them). It turns out that the approach is completely wooden and does not scale.

It was these two points that previously stopped scientists who tried to use Kolmogorov-Arnold in ML. Yes, yes, the idea is not new, but it was only really developed now: unlike its predecessors, the authors of KAN figured out how to get around the problems, and in the end got a brilliant result. So let's see what they did.

Naive KAN architecture

At first, the researchers, like other scientists before them, tried to use the theorem from the previous section head-on. Since we only need to find functions, in this case we end up with a neural network that does not have linear weights and activation functions in neurons at all. Here everything is the other way around. Instead of weights on the edges of the network, we train functions, and in neurons we simply add them.

Here is an example: for a network with two (n=2) input parameters, we get a two-layer (*since the composition depth in the theorem is two*) neural network with five (*since the theorem involves 2*n+1 = 5 functions*) neurons on the hidden layer.



"What about the fact that functions can be unteachable?" - you ask. Well, firstly, the authors substantiated that in the case of predicting dependencies from the real world, the appearance of such functions is extremely unlikely. Secondly, to get around this problem, in KAN we do not look for just any functions, but parametrize them with splines.

A spline is a smooth curve, a piecewise polynomial function, which is specified by different polynomials on different segments. Each spline is approximated using a specified number of points. The more points, the more accurate the approximation.

Splines are continuous and differentiable, which means that such an architecture can be easily trained using the familiar backpropagation method.

Generalized architecture

Unlike the differentiability problem, which scientists elegantly solved with splines, the KAN scalability problem did not give up so easily. How to make it possible to add more layers and neurons to the network? After all, this requires a generalized Kolmogorov-Arnold theorem, but it simply does not exist.

This is where the breakthrough part of the work lies. The researchers noticed that, by analogy with a perceptron, we can build a matrix of learnable objects on each layer. It's just that in our case these will not be parameters (numbers), but functions. In terms of the matrix, the original formula turns out to be not a law, but simply a special case of KAN with two layers. And the generalized KAN is a deeper composition of the following matrices:

$$\mathrm{KAN}(\mathbf{x}) = (\mathbf{\Phi}_{L-1} \circ \mathbf{\Phi}_{L-2} \circ \cdots \circ \mathbf{\Phi}_1 \circ \mathbf{\Phi}_0)\mathbf{x}.$$

And the theorem itself for KAN can be rewritten like this:

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

Otherwise, apart from the elegant internal mathematics, you can work with KAN the same way as with regular networks: add and remove neurons, stack layers, use dropout and even regularization.

Comparison with perceptron

Moving activations to the edges, although it does not seem like a global change, still brings with it a lot of changes. Here are the key aspects that differentiate KAN from a perceptron:

- Since approximation of each spline requires several points (even if we have K such points), KAN requires K times more parameters than MLP with the same depth and number of neurons on layers.
- Fortunately, the problem from the first point is mitigated by the fact that KAN requires many times fewer neurons to achieve MLP accuracy. Researchers have also empirically proven that KAN generalizes data much better.
- Due to the fact that in KAN we train functions, not numbers, we can increase the accuracy of the network without retraining it from scratch. In MLP, to achieve better accuracy, we can increase the number of layers and neurons, but this requires full retraining and, in fact, does not always work. In KAN, you simply need to add more points to the approximation grid. This guarantees the best result, and there is no need to retrain the neural network.
- KAN is more interpretable than MLP. But interpretability is one of the main problems of modern neural networks.
- KAN is better at approximating complex mathematical functions, so it has what you might call a "technical mind." The article shows that KAN solves differential equations an order of magnitude better and can (re)discover the laws of physics and mathematics.
- The architecture has a bottleneck: KAN learns about 10 times slower than MLP. Perhaps this will become a serious stumbling block, or perhaps engineers will quickly learn to optimize the efficiency of such networks.

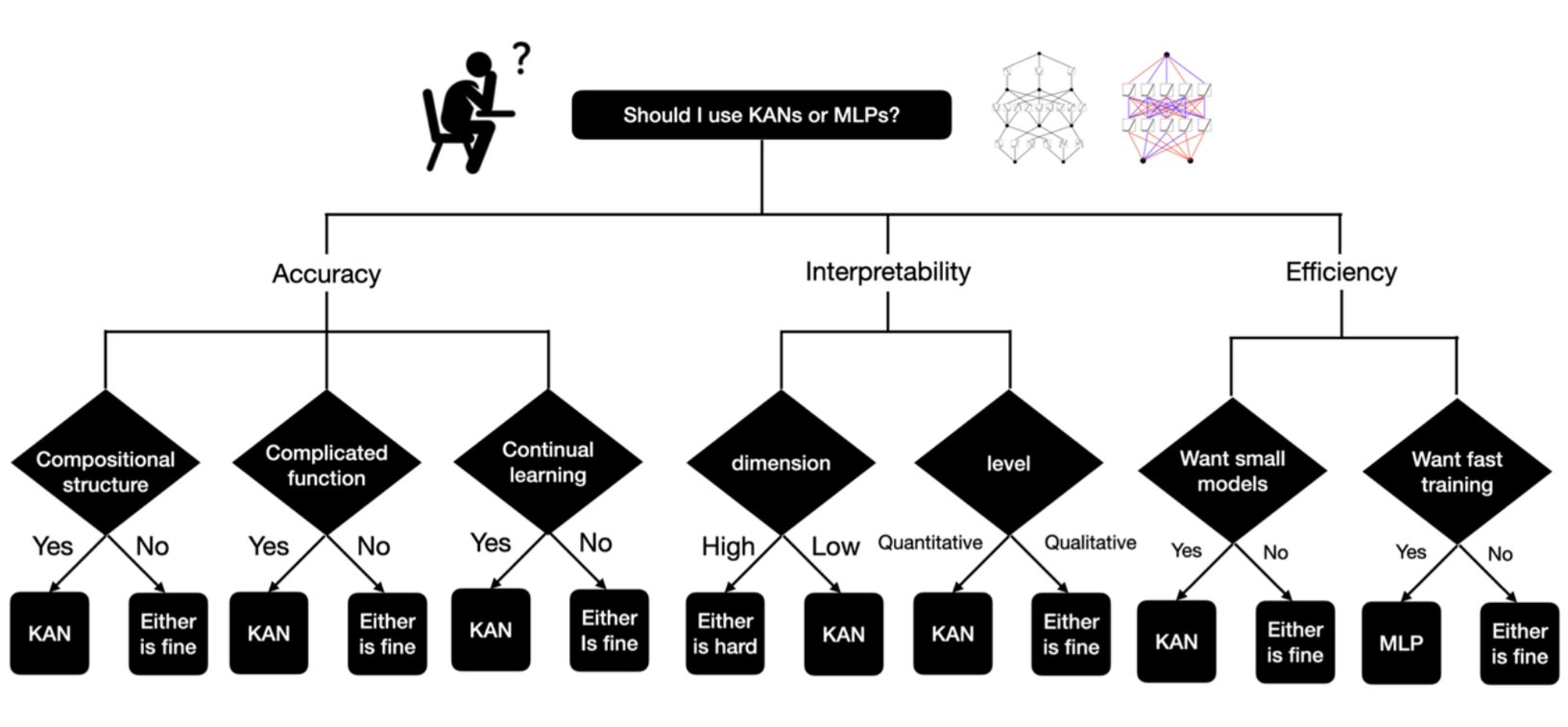


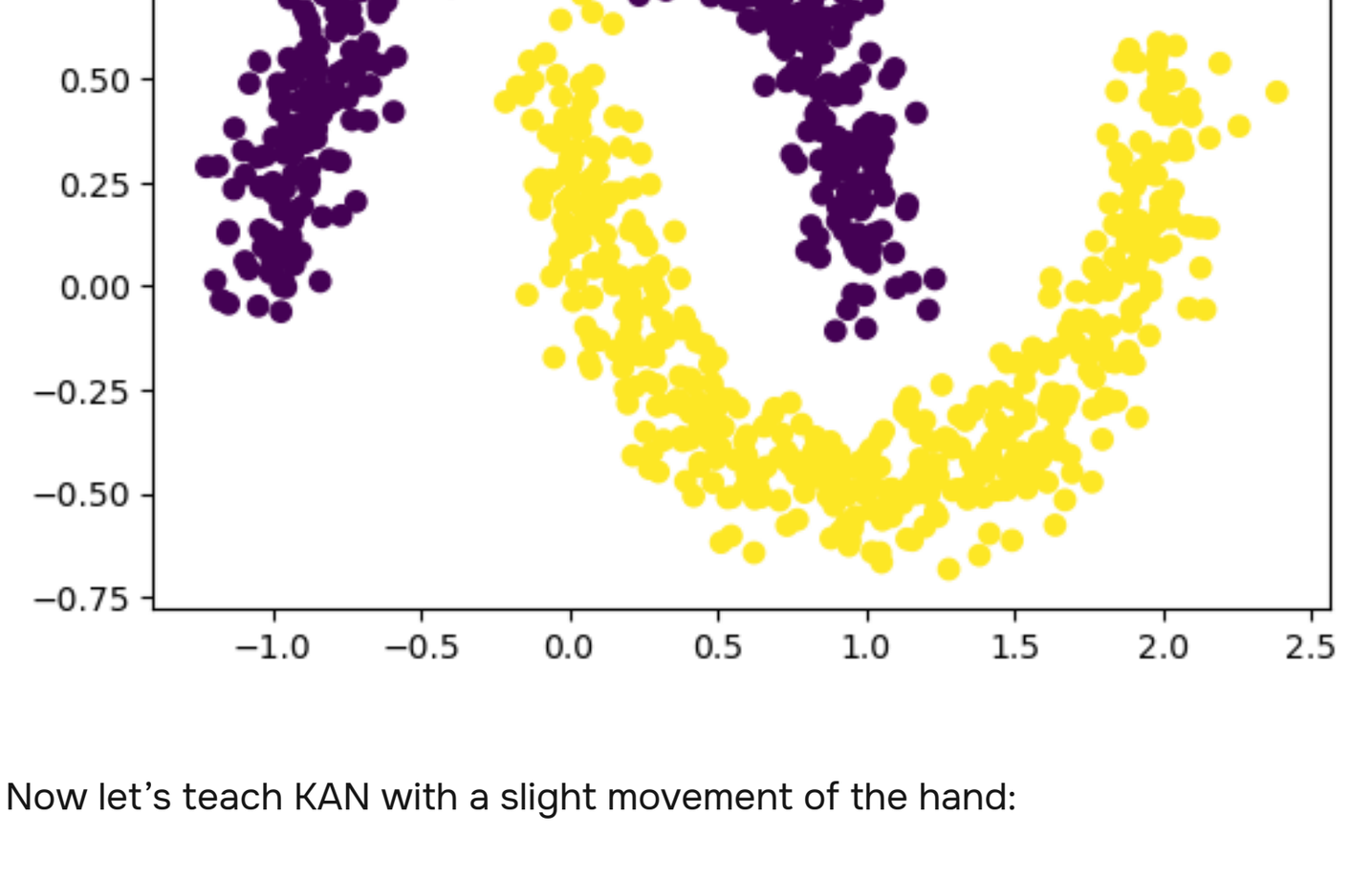
Figure 6.1: Should I use KANs or MLPs?

Code!

The article is not everything. The researchers also posted the code, and even released a library with which you can play with KAN out of the box, it's called **pykan** . Documentation for it can be found **here**
.

Let's take an example of how to train KAN for a classification task. First, let's generate a dataset:

```
from kan import KAN import matplotlib . pyplot as plt from sklearn . datasets import
make_moons import torch import numpy as np dataset = { } train_input , train_label =
make_moons ( n_samples = 1000 , shuffle = True , noise = 0.1 , random_state = None )
test_input , test_label = make_moons ( n_samples = 1000 , shuffle = True , noise =
0.1 , random_state = None ) dataset [ 'train_input' ] = torch . from_numpy (
train_input ) dataset [ 'test_input' ] = torch . from_numpy ( test_input ) dataset [
'train_label' ] = torch . from_numpy ( train_label [ : , None ] ) dataset [
'test_label' ] = torch . from_numpy ( test_label [ : , None ] ) X = dataset [
'train_input' ] y = dataset [ 'train_label' ] plt . scatter ( X [ : , 0 ] , X [ : , 1
] , c = y [ : , 0 ] )
```



Now let's teach KAN with a slight movement of the hand:

```
model = KAN ( width = [ 2 , 1 ] , grid = 3 , k = 3 ) def train_acc ( ) : return torch
. mean ( ( torch . round ( model ( dataset [ 'train_input' ] ) [ : , 0 ] ) == dataset
[ 'train_label' ] [ : , 0 ] ) . float ( ) ) def test_acc ( ) : return torch . mean (
( torch . round ( model ( dataset [ 'test_input' ] ) [ : , 0 ] ) == dataset [
'test_label' ] [ : , 0 ] ) . float ( ) ) results = model . train ( dataset , opt =
"LBFGS" , steps = 20 , metrics = ( train_acc , test_acc ) ) ;
```

The accuracy in this example will be one on the training and test samples.

By the way, the project repository contains very beautiful and understandable **notebooks** , in which you can find tutorials on the library and cases of using KAN.

Conclusion

Is KAN the new era of deep learning? There is no exact answer, but the method has every chance. At the very least, this is a big impetus for research. We will wait for news about opportunities to improve existing models.