

# Kolmogorov– Arnold Networks

## Umar Jamil

Downloaded from: <https://github.com/hkproj/kan-notes>

License: Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0):

<https://creativecommons.org/licenses/by-nc/4.0/legalcode>

**Not for commercial use**

# Topics

- Review of Multilayer Perceptron
- Introduction to data fitting
- Bézier Curves
- B-Splines
- Universal Approximation Theorem
- Kolmogorov-Arnold Representation Theorem
- MLPs vs KAN
- Properties
  - Multi-layer KANs
  - Parameters count: MLPs vs KANs
  - Grid extension
  - Interpretability
  - Continual training

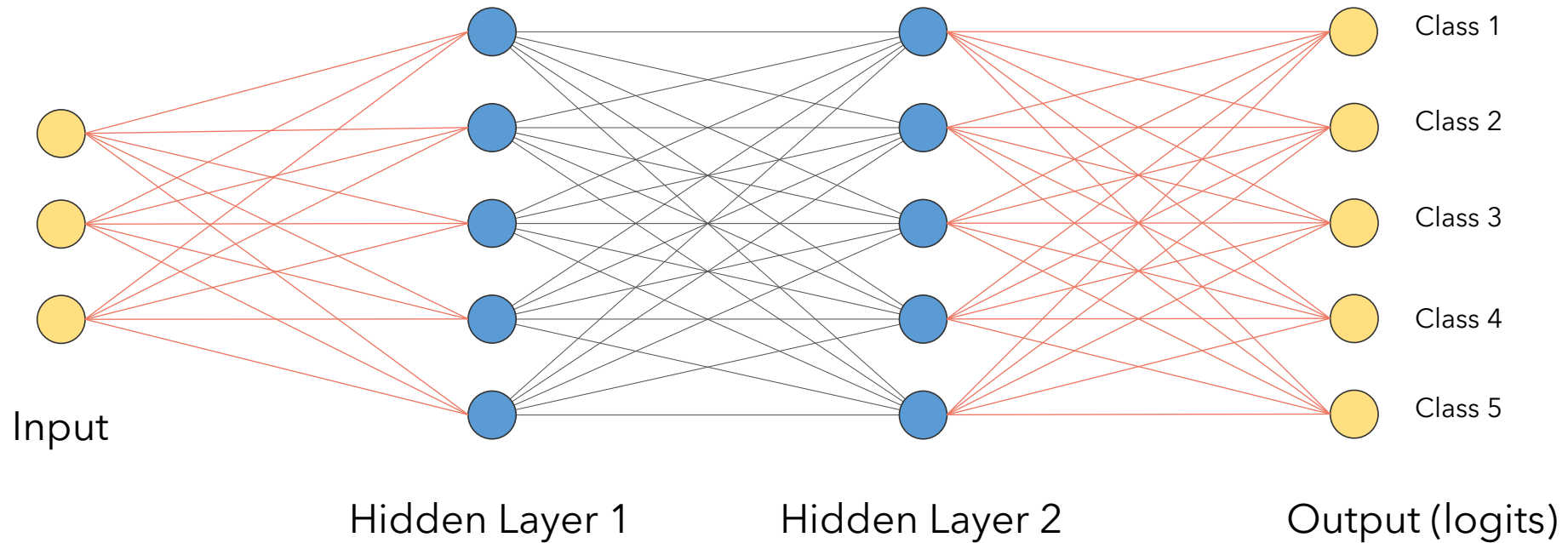
# Prerequisites

- Basics of calculus (derivative)
- Basics of deep learning (backpropagation)

# The Multi-layer Perceptron (MLP)

A multilayer perceptron is a neural network made up of multiple layers of neurons, organized in a feed-forward way, with nonlinear activation functions in between.

**How does it work?**



# The Linear layer in PyTorch

## Linear

---

CLASS `torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [\[SOURCE\]](#)

Applies a linear transformation to the incoming data:  $y = xA^T + b$ .

# The Linear layer in detail

A linear layer in a MLP is made of a weight matrix and a bias matrix.

$$z_1 = (r_1 + b_1) = (\sum_{i=1}^3 a_i w_i + b_1)$$

**b** =  
(1, 5)

n1	n2	n3	n4	n5
b				
1				

The bias vector will be broadcasted to every row in the  $XW^T$  table.

+

**X** =  
(10, 3)

	f1	f2	f3
Item 1	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>
Item 2			
Item 3			
Item 10			

**O** = **XW<sup>T</sup>** + **b**

**W<sup>T</sup>** =  
(3, 5)

	n1	n2	n3	n4	n5
w <sub>1</sub>					
w <sub>2</sub>					
w <sub>3</sub>					

**XW<sup>T</sup>** =  
(10, 5)

	f1	f2	f3	f4	f5
Item 1	r <sub>1</sub>				
Item 2					
Item 3					
Item 10					

**O** =  
(10, 5)

	f1	f2	f3	f4	f5
Item 1	z <sub>1</sub>				
Item 2					
Item 3					
Item 10					

# Why do we need activation functions?

After each Linear layer, we usually apply a nonlinear activation function. Why?

$$o_1 = xw_1^T + b_1$$

$$o_2 = (o_1)w_2^T + b_2$$

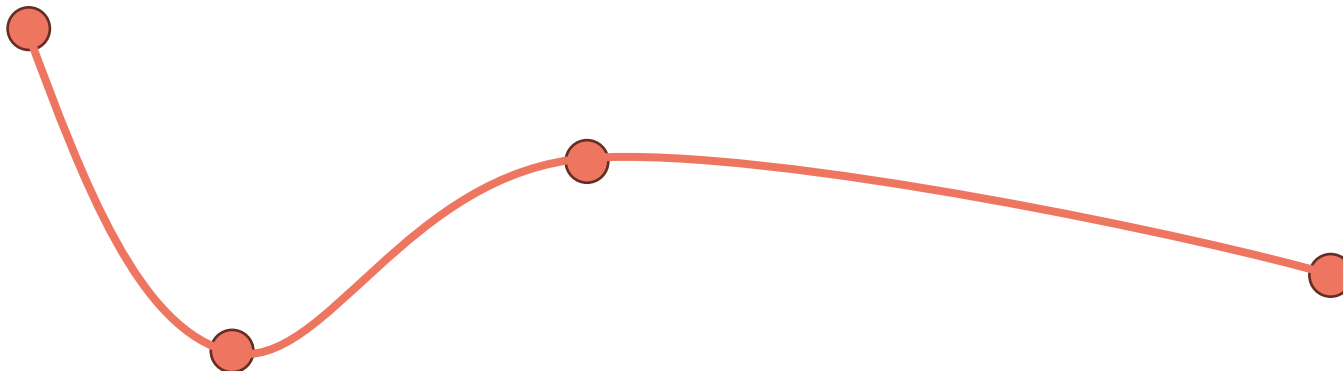
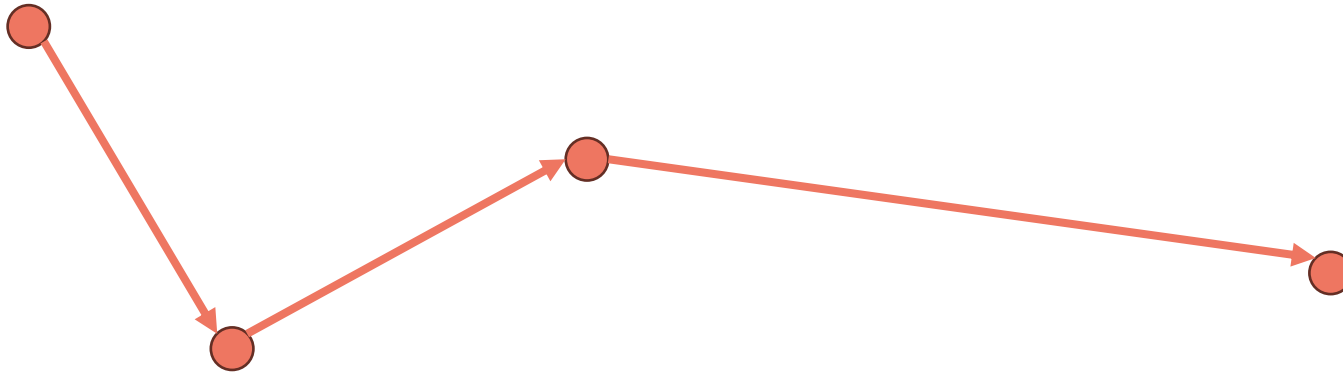
$$o_2 = (xw_1^T + b_1)w_2^T + b_2$$

$$o_2 = xw_1^Tw_2^T + b_1w_2^T + b_2$$

As you can see, if we do not apply any activation functions, the output will just be a linear combination of the inputs, which means that our MLP will not be able to learn any non-linear mapping between the input and output, which represents most of the real-world data.

# Introduction to data fitting

Imagine you're making a 2D game and you want animate your sprite (character) to pass through a series of points. One way would be to make a straight line from one point to the next, but that wouldn't look so good. What if you could create a smoother path, like the one below?



# Smooth curves through polynomial curves

How to find the equation of such a smooth curve?

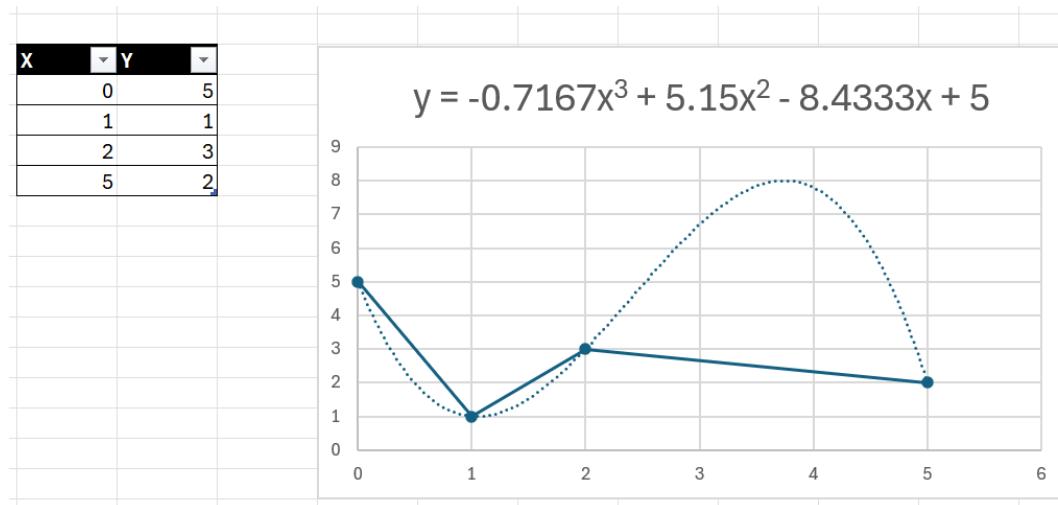
One way is to write the generic equation of a polynomial curve and force it to pass through the series of points to get the coefficients of the equation.

We have 4 points, so we can make a system of equations with 4 equations, which means we can solve for 4 variables: yes, we get a polynomial with degree 3.

$$y = ax^3 + bx^2 + cx + d$$

We can write our system of equations as follows and solve to find the equation of the curve:

$$\begin{aligned} 5 &= a(0)^3 + b(0)^2 + c(0) + d \\ 1 &= a(1)^3 + b(1)^2 + c(1) + d \\ 3 &= a(2)^3 + b(2)^2 + c(2) + d \\ 2 &= a(5)^3 + b(5)^2 + c(5) + d \end{aligned}$$

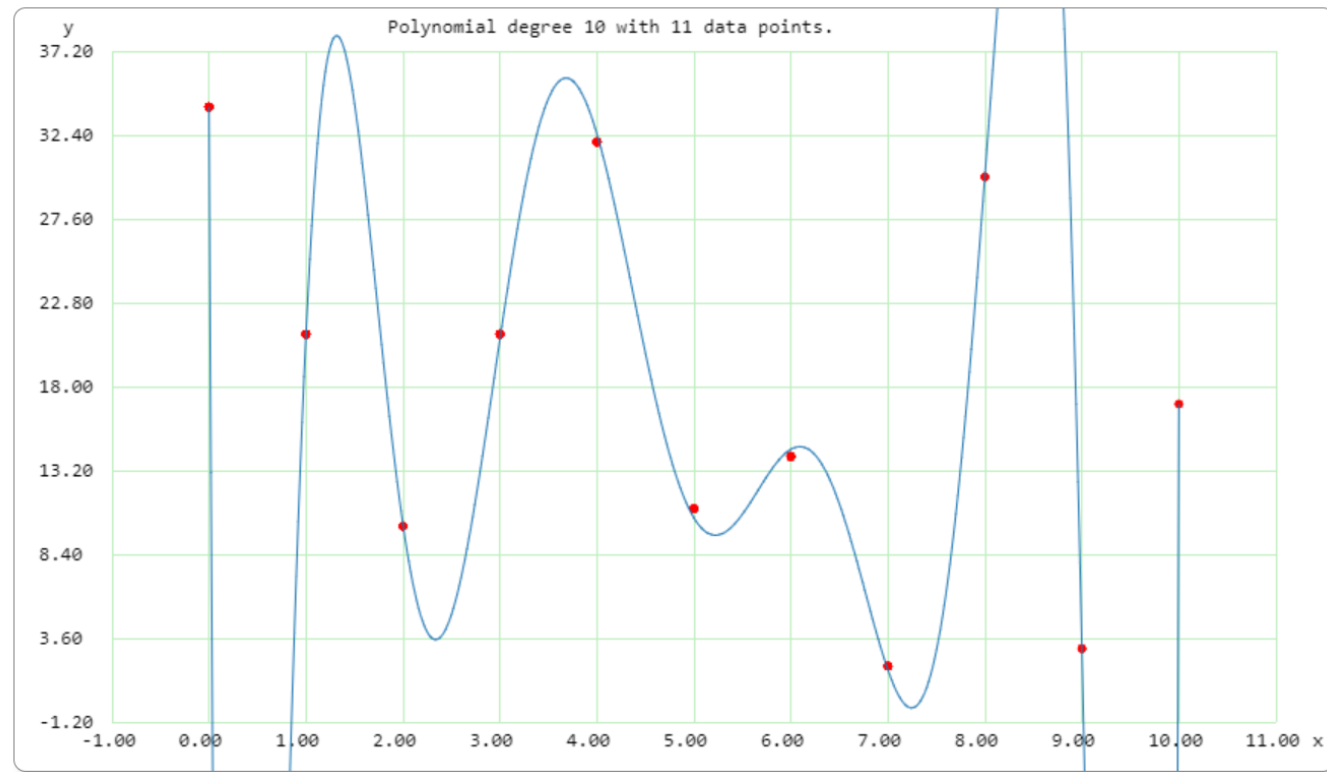




# What if I have hundreds of points?

If you have  $N$  points, you need a polynomial of degree  $N - 1$  if you want the line to pass through all those points. But as you can see, when we have lots of points, the polynomial starts getting crazy on the extremes. We wouldn't want the character in our 2D game to go out of the screen while we're animating it, right?

**Thankfully, someone took the time to solve this problem, because we have Bézier curves!**



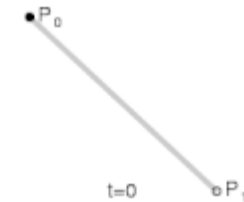
Source: <https://arachnoid.com/polysolve/>

# Bézier curves

A Bézier curve is a parametric curve (which means that all the coordinates of the curve depend on an independent variable  $t$ , between 0 and 1).

For example, given two points, we can calculate the linear B curve as the following interpolation:

$$B(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + tP_1$$

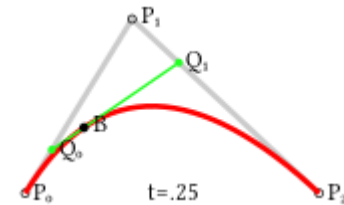


Given three points, we can calculate the quadratic Bézier curve that interpolates them.

$$Q_0(t) = (1 - t)P_0 + tP_1$$

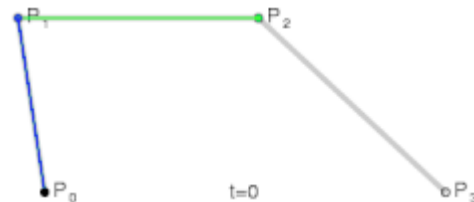
$$Q_1(t) = (1 - t)P_1 + tP_2$$

$$\begin{aligned} B(t) &= (1 - t)Q_0 + tQ_1 \\ &= (1 - t)[(1 - t)P_0 + tP_1] + t[(1 - t)P_1 + tP_2] \\ &= (1 - t)^2P_0 + 2(1 - t)tP_1 + t^2P_2 \end{aligned}$$



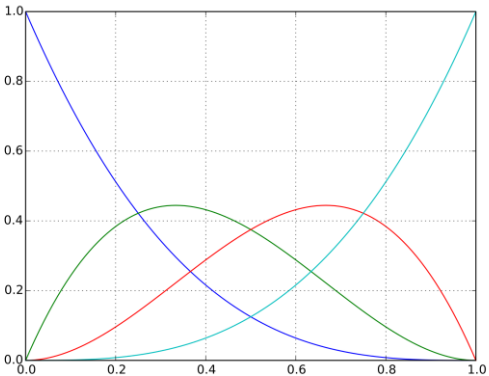
Source: [Wikipedia](https://en.wikipedia.org/wiki/B%C3%A9zier_curve)

With four points, we can proceed with a similar reasoning.



# Bézier curves: going deeper

Yes, we can go deeper! If we have  $n + 1$  points, we can find the  $n$  degree Bézier curve using the following formula



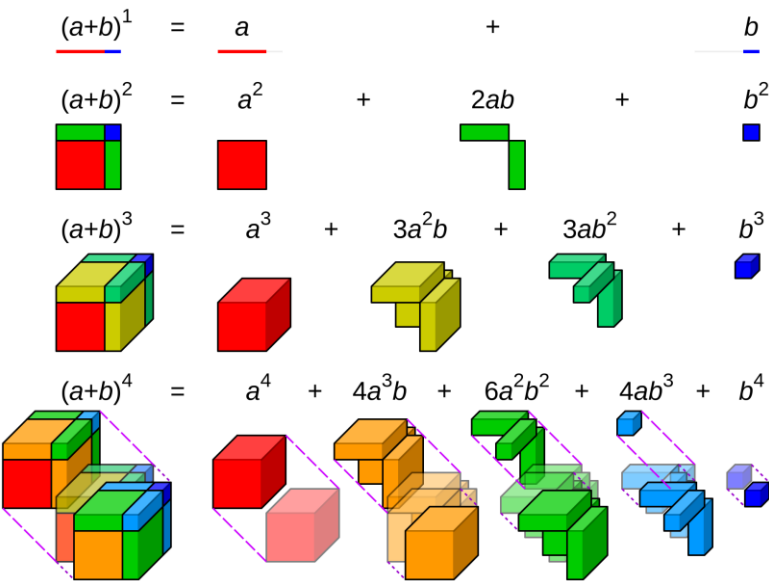
**Blue:**  $b_{0,3}(t)$   
**Green:**  $b_{1,3}(t)$   
**Red:**  $b_{2,3}(t)$   
**Cyan:**  $b_{3,3}(t)$

**Binomial coefficients**

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

$$B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i = \sum_{i=0}^n b_{i,n}(t) P_i$$

*Bernstein basis polynomials*

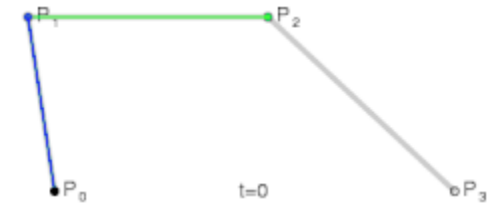
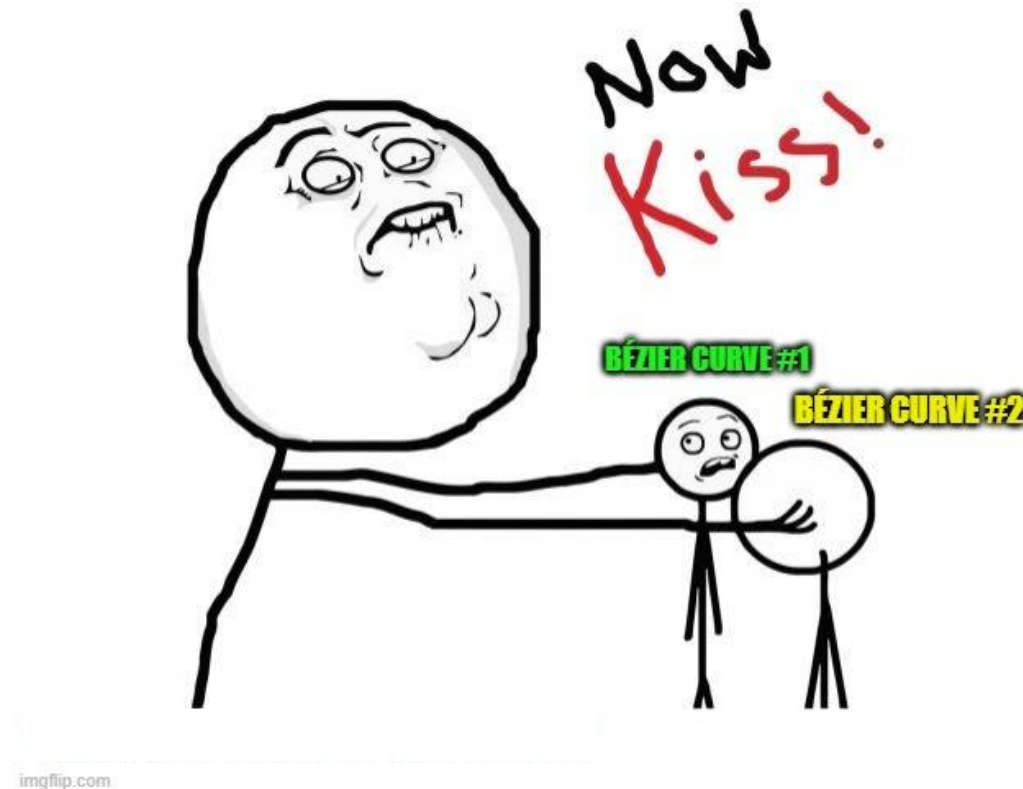


Source: [Wikipedia](https://en.wikipedia.org/wiki/Bernstein_polynomial)

# From Bézier curves to B-Splines

If you have lots of points (say  $n$ ), you need a Bézier curve with a degree  $n-1$  to approximate it well, but that can be quite complicated computationally to calculate.

Someone wise thought: why don't we stitch together many Bézier curves between all these points, instead of one big Bézier curve that interpolates all of them?



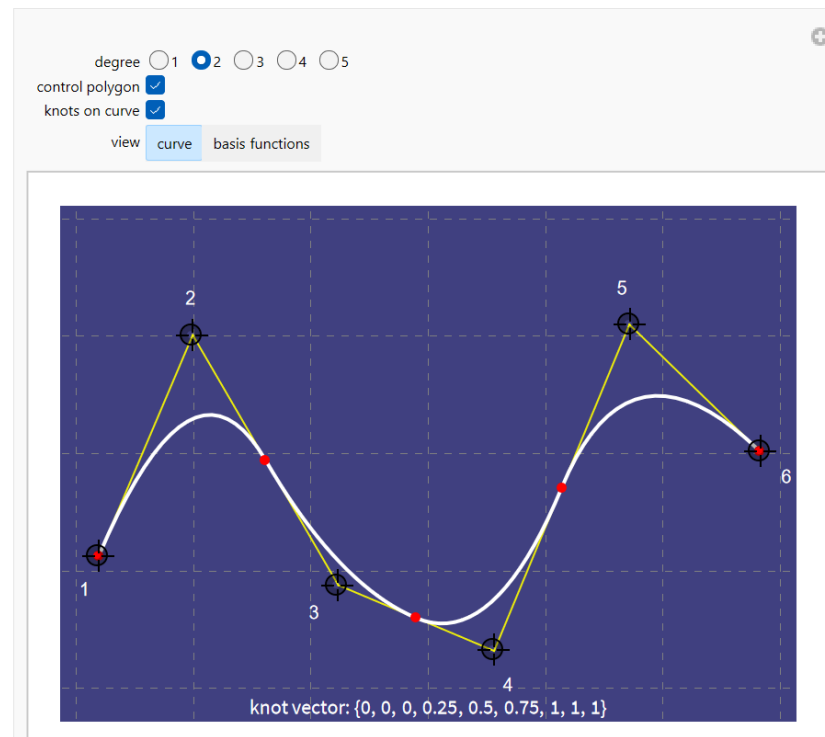
Source: [Wikipedia](https://en.wikipedia.org/wiki/B-spline)

# B-splines in detail

A  $k$ -degree B-Spline curve that is defined by  $n$  control points, will consist of  $n - k$  Bézier curves.  
For example, if we want to use a quadratic Bézier curve and we have 6 points, we need  $6 - 2 = 4$  Bézier curves.

**In this case we have  $n=6$  and  $k=2$**

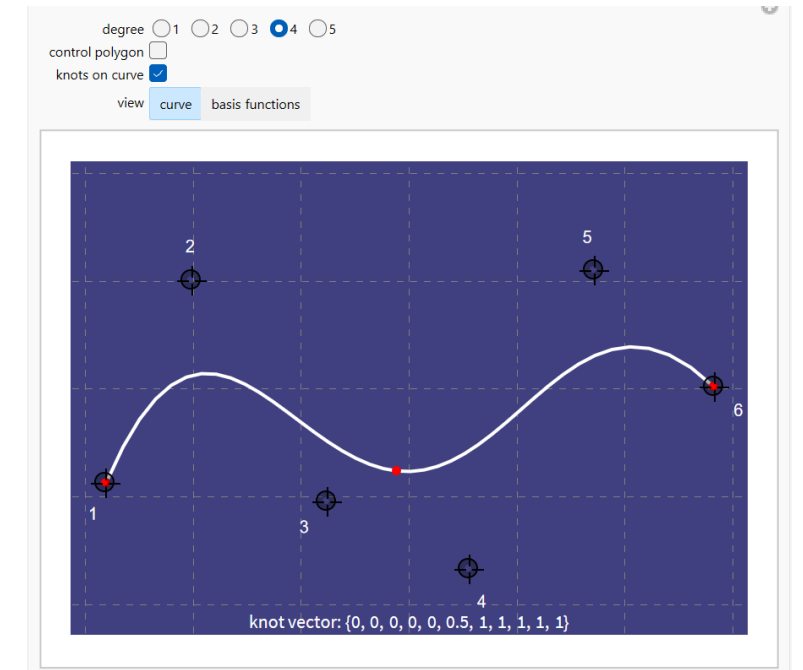
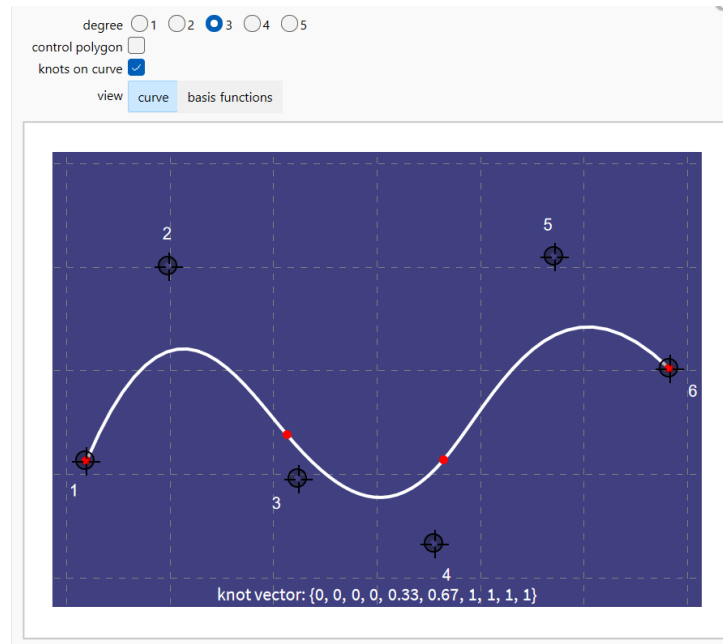
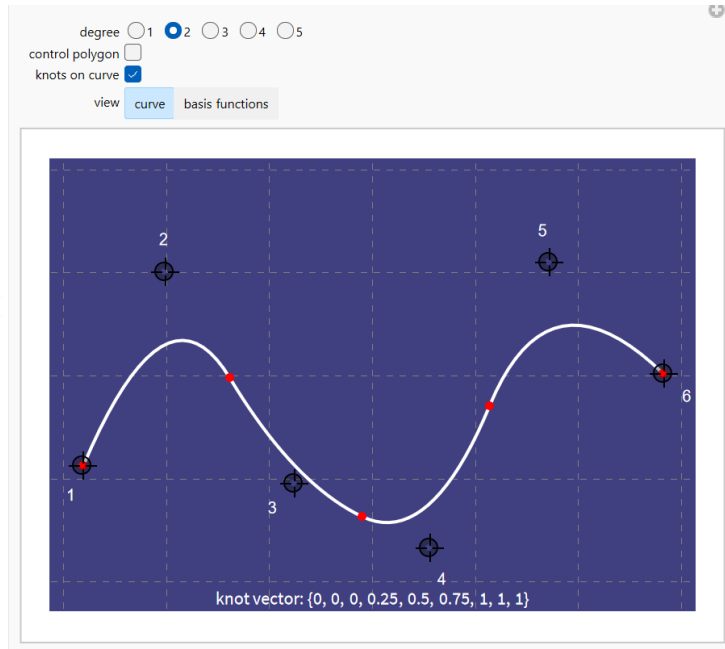
## B-Spline Curve with Knots



Source: [Wolfram Alpha](https://www.wolframalpha.com/)

# B-splines in detail

The degree of our B-Spline also tells what kind of continuity we get.



## 1.4.1 B-splines

An order  $k$  B-spline is formed by joining several pieces of polynomials of degree  $k - 1$  with at most  $C^{k-2}$  continuity at the breakpoints.

Source: [MIT](https://www.mit.edu/~dave/teaching/6.037/lectures/14.1/B-splines.pdf)

# Calculating B-splines: algorithm

A B-spline curve is defined as a linear combination of control points  $\mathbf{p}_i$  and B-spline basis functions  $N_{i,k}(t)$  given by

$$\mathbf{r}(t) = \sum_{i=0}^n \mathbf{p}_i N_{i,k}(t), \quad n \geq k - 1, \quad t \in [t_{k-1}, t_{n+1}] . \quad (1.62)$$

In this context the control points are called *de Boor points*. The basis function  $N_{i,k}(t)$  is defined on a *knot vector*

$$\mathbf{T} = (t_0, t_1, \dots, t_{k-1}, t_k, t_{k+1}, \dots, t_{n-1}, t_n, t_{n+1}, \dots, t_{n+k}) , \quad (1.63)$$

where there are  $n + k + 1$  elements, i.e. the number of control points  $n + 1$  plus the order of the curve  $k$ . Each knot *span*  $t_i \leq t \leq t_{i+1}$  is mapped onto a polynomial curve between two successive joints  $\mathbf{r}(t_i)$  and  $\mathbf{r}(t_{i+1})$ . Normalization of the knot vector, so it covers the interval  $[0, 1]$ , is helpful in improving numerical accuracy in floating point arithmetic computation due to the higher density of floating point numbers in this interval [133,300].

Given a knot vector  $\mathbf{T}$ , the associated B-spline basis functions,  $N_{i,k}(t)$ , are defined as:

$$N_{i,1}(t) = \begin{cases} 1 & \text{for } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} , \end{cases} \quad (1.58)$$

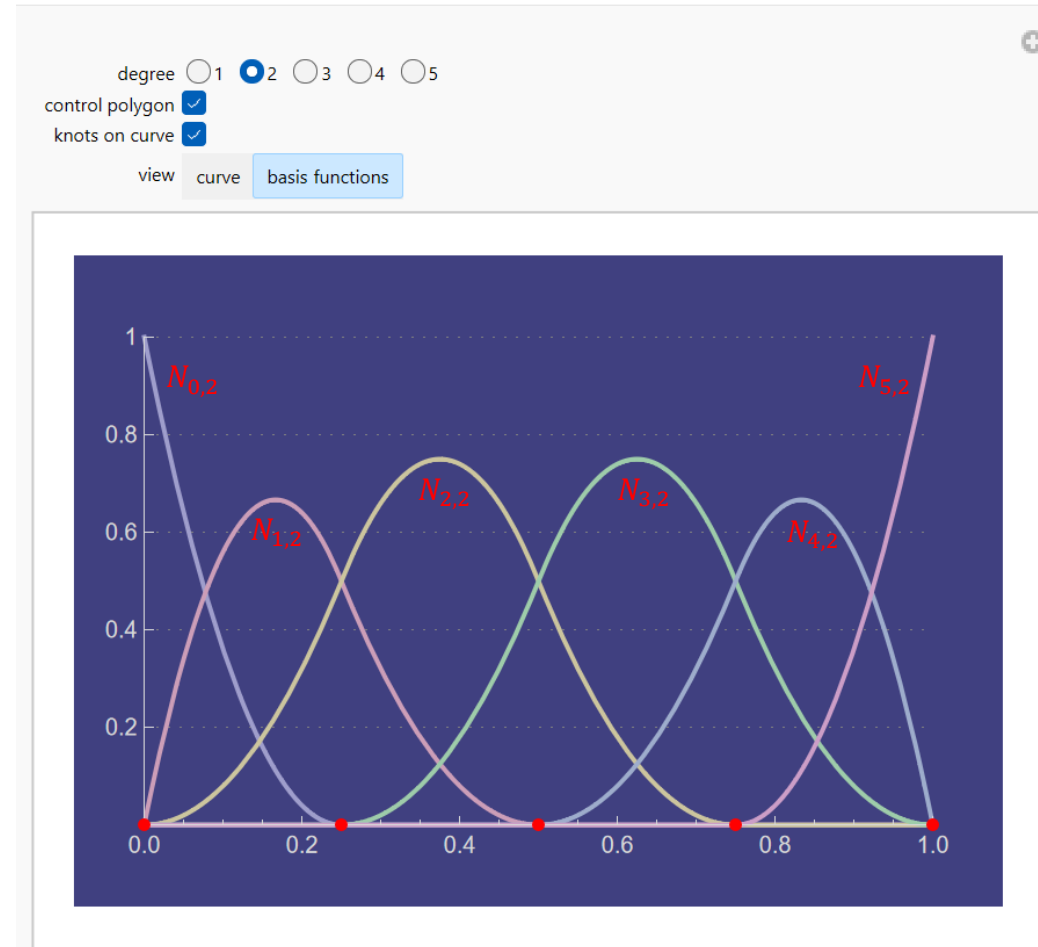
for  $k = 1$ , and

$$N_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t) , \quad (1.59)$$

for  $k > 1$  and  $i = 0, 1, \dots, n$ . These equations have the following properties [175]:

Source: [MIT](#)

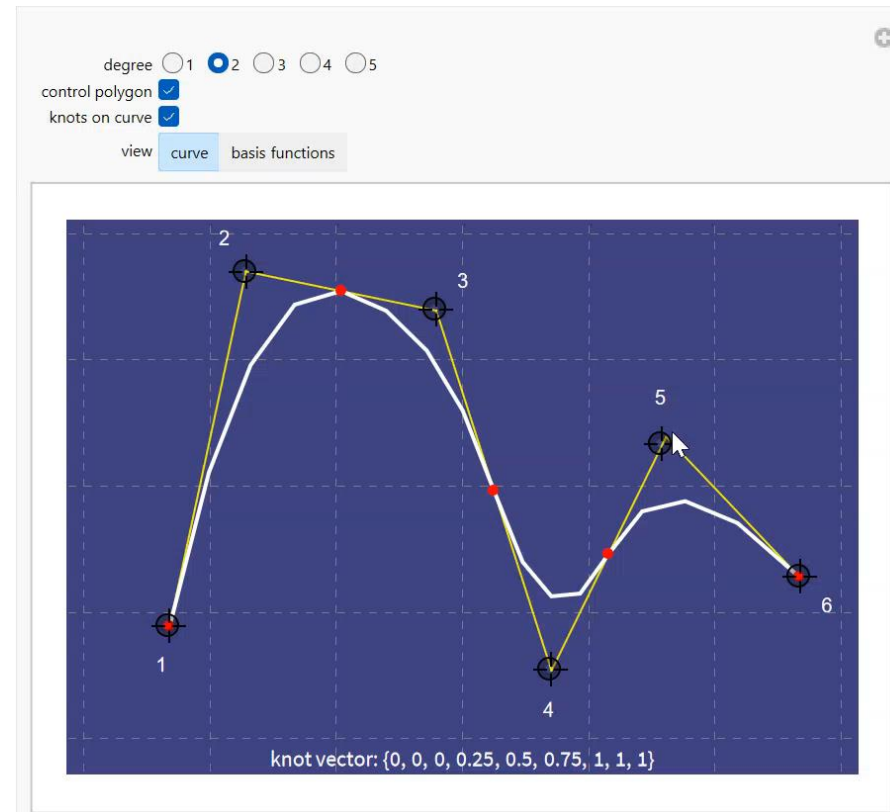
# B-Splines: basis functions





# B-splines: local control

Moving a control point only changes the curve locally (in the proximity of the control point), leaving the adjacent Bezier curves unchanged!



# Universal Approximation Theorem

We can think of neural networks as functions approximators. Usually, we have access to some data points generated by an ideal function that we do not have access. The goal of training a neural network is to approximate this ideal function (that we do not have access).

But how do we know if a neural network is powerful enough to model our ideal function? What can we say about the *expressive power* of neural networks?

This is what the universal approximation theorem is all about: it is a series of results that put limits on what neural networks can learn.

It has been proven that neural networks with a certain width (number of neurons) and depth (number of layers) can approximate any continuous function if using specific non-linear activation functions, for example the ReLU function. Check [Wikipedia](#) for more theoretical results.

I want to emphasize what it means to be a universal approximator: it means that given an ideal function (or a family of functions) that models the training data, the network can learn to approximate it as good as we want, that is, given an error  $\epsilon$ , we can always find an approximate function that is close to the ideal function within this error limit.

This is however a theoretical result; it doesn't tell us how to do it practically. On a practical level, we have many problems:

- Achieving good approximations may take enormous amounts of computational power
- We may need a large big quantity of training data
- Our hardware may not be able to represent certain weights in 32 bit
- Our optimizer may remain stuck in a local minima

So as you can see, just because a neural network can learn anything, doesn't mean we are be able to learn it in practice. But at least we know that the limits are practical.

# Kolmogorov-Arnold representation theorem

## Kolmogorov-Arnold representation theorem

Kolmogorov-Arnold representation theorem states that if  $f$  is a multivariate continuous function on a bounded domain, then it can be written as a finite composition of continuous functions of a single variable and the binary operation of addition. More specifically, for a smooth  $f : [0, 1]^n \rightarrow \mathbb{R}$ ,

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

where  $\phi_{q,p} : [0, 1] \rightarrow \mathbb{R}$  and  $\Phi_q : \mathbb{R} \rightarrow \mathbb{R}$ . In a sense, they showed that the only true multivariate function is addition, since every other function can be written using univariate functions and sum.

# Kolmogorov-Arnold Networks

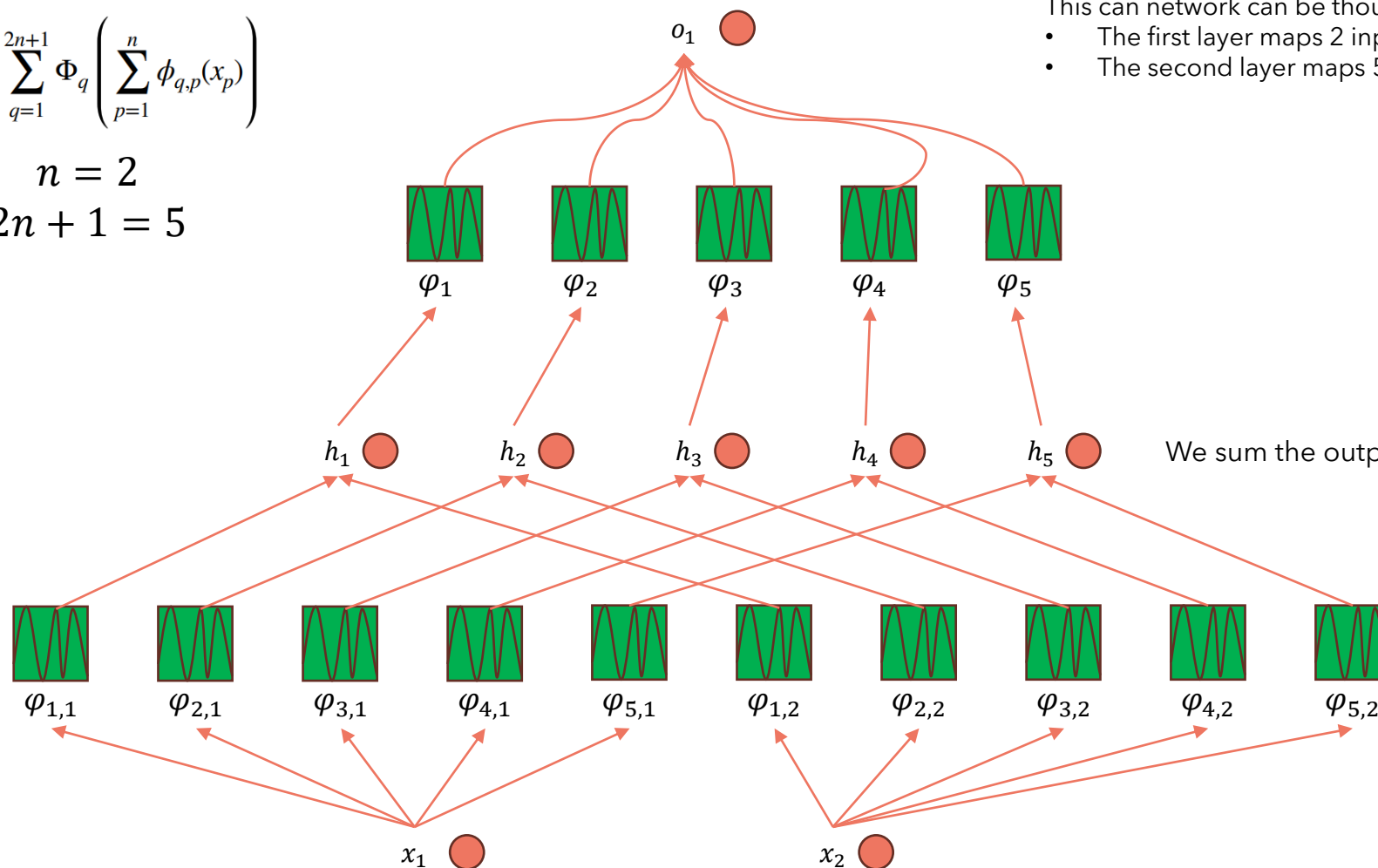
$$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left( \sum_{p=1}^n \phi_{q,p}(x_p) \right)$$

$$n = 2$$

$$2n + 1 = 5$$

This network can be thought of as two layers applied in sequence:

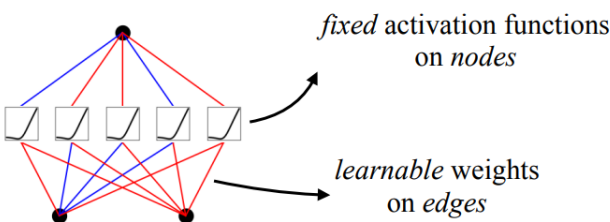
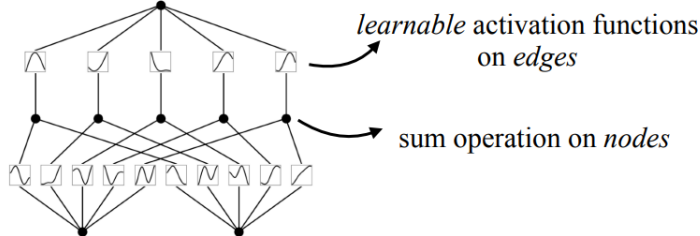
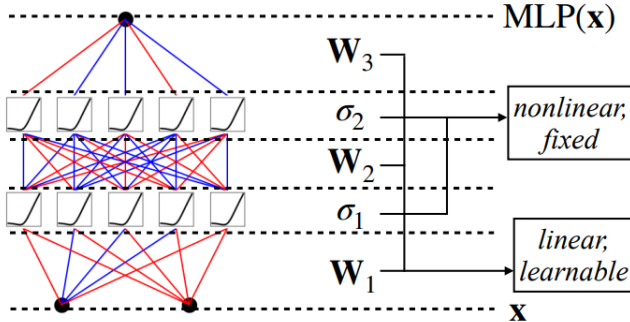
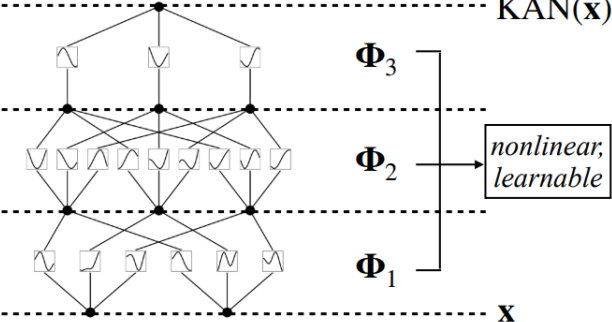
- The first layer maps 2 input features into 5 output features.
- The second layer maps 5 input features into 1 output feature.



We sum the output of the learnable functions

Instead of having learnable weights, we have **learnable functions**

# MLP vs KAN

Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left( \sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	<p>(a)</p> 	<p>(b)</p> 
Formula (Deep)	$\text{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\text{KAN}(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	<p>(c)</p> 	<p>(d)</p> 

# Multi-layer KAN

The breakthrough occurs when we notice the analogy between MLPs and KANs. In MLPs, once we define a layer (which is composed of a linear transformation and nonlinearities), we can stack more layers to make the network deeper. To build deep KANs, we should first answer: “what is a KAN layer?” It turns out that a KAN layer with  $n_{\text{in}}$ -dimensional inputs and  $n_{\text{out}}$ -dimensional outputs can be defined as a matrix of 1D functions

$$\Phi = \{\phi_{q,p}\}, \quad p = 1, 2, \dots, n_{\text{in}}, \quad q = 1, 2, \dots, n_{\text{out}}, \quad (2.2)$$

where the functions  $\phi_{q,p}$  have trainable parameters, as detailed below. In the Kolmogorov-Arnold theorem, the inner functions form a KAN layer with  $n_{\text{in}} = n$  and  $n_{\text{out}} = 2n + 1$ , and the outer functions form a KAN layer with  $n_{\text{in}} = 2n + 1$  and  $n_{\text{out}} = 1$ . So the Kolmogorov-Arnold representations in Eq. (2.1) are simply compositions of two KAN layers. Now it becomes clear what it means to have deeper Kolmogorov-Arnold representations: simply stack more KAN layers!

## Layer 2

5 input features, 1 output features  
total of 5 functions to “learn”

## Layer 1

2 input features, 5 output features  
total of 10 functions to “learn”

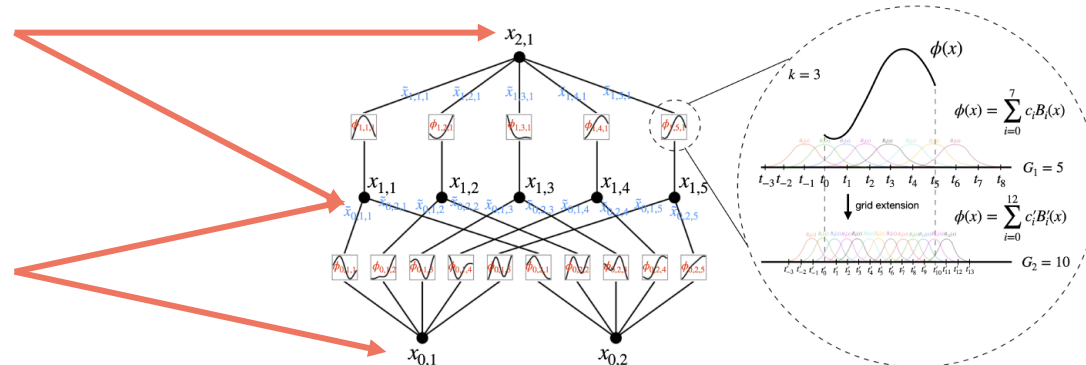


Figure 2.2: Left: Notations of activations that flow through the network. Right: an activation function is parameterized as a B-spline, which allows switching between coarse-grained and fine-grained grids.

# Implementation details

**Implementation details.** Although a KAN layer Eq. (2.5) looks extremely simple, it is non-trivial to make it well optimizable. The key tricks are:

- (1) Residual activation functions. We include a basis function  $b(x)$  (similar to residual connections) such that the activation function  $\phi(x)$  is the sum of the basis function  $b(x)$  and the spline function:

$$\phi(x) = w(b(x) + \text{spline}(x)). \quad (2.10)$$

We set

$$b(x) = \text{silu}(x) = x/(1 + e^{-x}) \quad (2.11)$$

in most cases.  $\text{spline}(x)$  is parametrized as a linear combination of B-splines such that

$$\text{spline}(x) = \sum_i c_i B_i(x) \quad (2.12)$$

where  $c_i$ s are trainable. In principle  $w$  is redundant since it can be absorbed into  $b(x)$  and  $\text{spline}(x)$ . However, we still include this  $w$  factor to better control the overall magnitude of the activation function.

- (2) Initialization scales. Each activation function is initialized to have  $\text{spline}(x) \approx 0$ <sup>2</sup>.  $w$  is initialized according to the Xavier initialization, which has been used to initialize linear layers in MLPs.
- (3) Update of spline grids. We update each grid on the fly according to its input activations, to address the issue that splines are defined on bounded regions but activation values can evolve out of the fixed region during training<sup>3</sup>.

# Parameters count

Then there are in total  $O(N^2L(G + k)) \sim O(N^2LG)$  parameters. In contrast, an MLP with depth  $L$  and width  $N$  only needs  $O(N^2L)$  parameters, which appears to be more efficient than KAN. Fortunately, KANs usually require much smaller  $N$  than MLPs, which not only saves parameters, but also achieves better generalization (see e.g., Figure 3.1 and 3.3) and facilitates interpretability. We remark that for 1D problems, we can take  $N = L = 1$  and the KAN network in our implementation is nothing but a spline approximation. For higher dimensions, we characterize the generalization behavior of KANs with a theorem below.

Compared to MLP, we also have  $(G+k)$  parameters for each activation, because we need to learn where to put the control points for the B-Splines.

$$\mathbf{r}(t) = \sum_{i=0}^n \mathbf{p}_i N_{i,k}(t), \quad n \geq k-1, \quad t \in [t_{k-1}, t_{n+1}] . \quad (1.62)$$

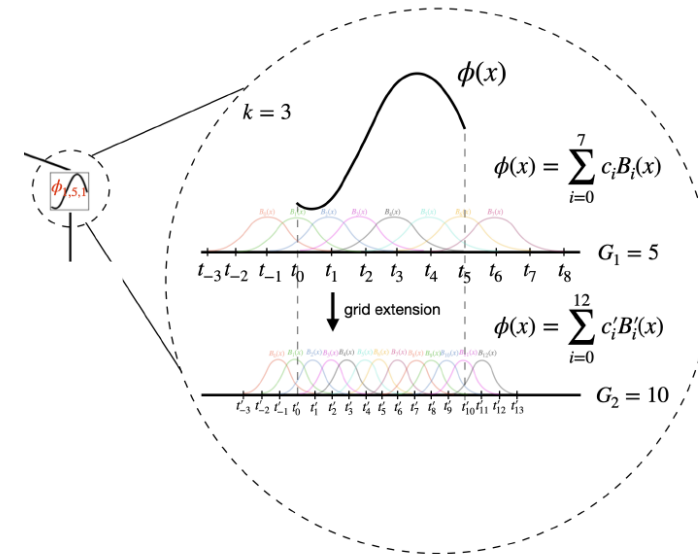


# Grid extension

We can increase the number of “control points” in the B-Spline to give it more “degrees of freedom” to better approximate more complex functions, meaning that we can extend the grid of an existing pre-trained network.

The parameters  $c'_j$ s can be initialized from the parameters  $c_i$  by minimizing the distance between  $f_{\text{fine}}(x)$  to  $f_{\text{coarse}}(x)$  (over some distribution of  $x$ ):

$$\{c'_j\} = \underset{\{c'_j\}}{\operatorname{argmin}} \mathbb{E}_{x \sim p(x)} \left( \sum_{j=0}^{G_2+k-1} c'_j B'_j(x) - \sum_{i=0}^{G_1+k-1} c_i B_i(x) \right)^2, \quad (2.16)$$



# Interpretability

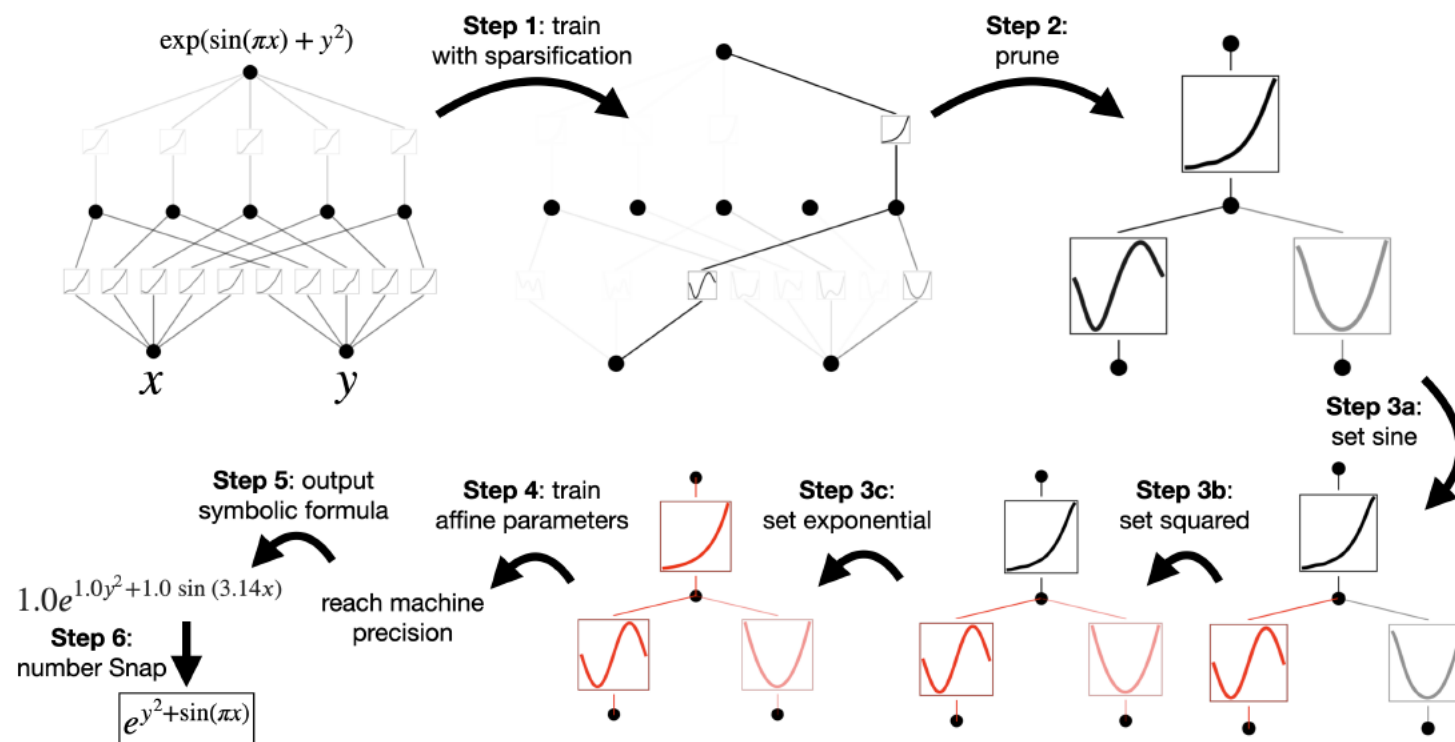


Figure 2.4: An example of how to do symbolic regression with KAN.

# Continual learning

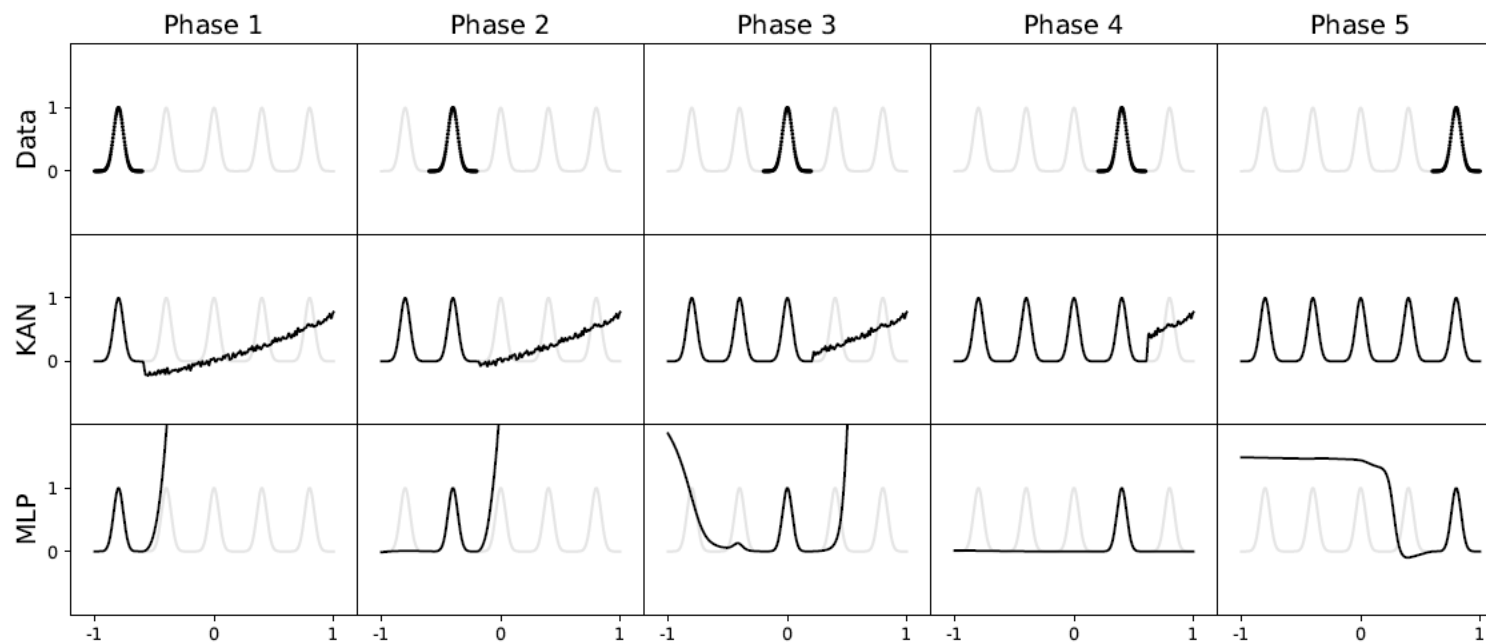


Figure 3.4: A toy continual learning problem. The dataset is a 1D regression task with 5 Gaussian peaks (top row). Data around each peak is presented sequentially (instead of all at once) to KANs and MLPs. KANs (middle row) can perfectly avoid catastrophic forgetting, while MLPs (bottom row) display severe catastrophic forgetting.

Thanks for watching!  
Don't forget to subscribe for  
more amazing content on AI  
and Machine Learning!