



دانشگاه صنعتی امیرکبیر  
(پلی تکنیک تهران)  
دانشکده ریاضی و علوم کامپیوتر

مباحثی در علوم کامپیوتر

گزارش ۴:

پیاده سازی شبکه عصبی KAN

نگارش

گروه ۱۱

استاد درس

دکتر فاطمه شاکری

آبان ۱۴۰۲

صفحه

## فهرست مطالب

گزارش ۴ 1

۱. فصل اول مقدمه..... 3

۲. فصل دوم پیاده سازی KAN با استفاده از Pytorch..... 22

منابع و مراجع..... 34

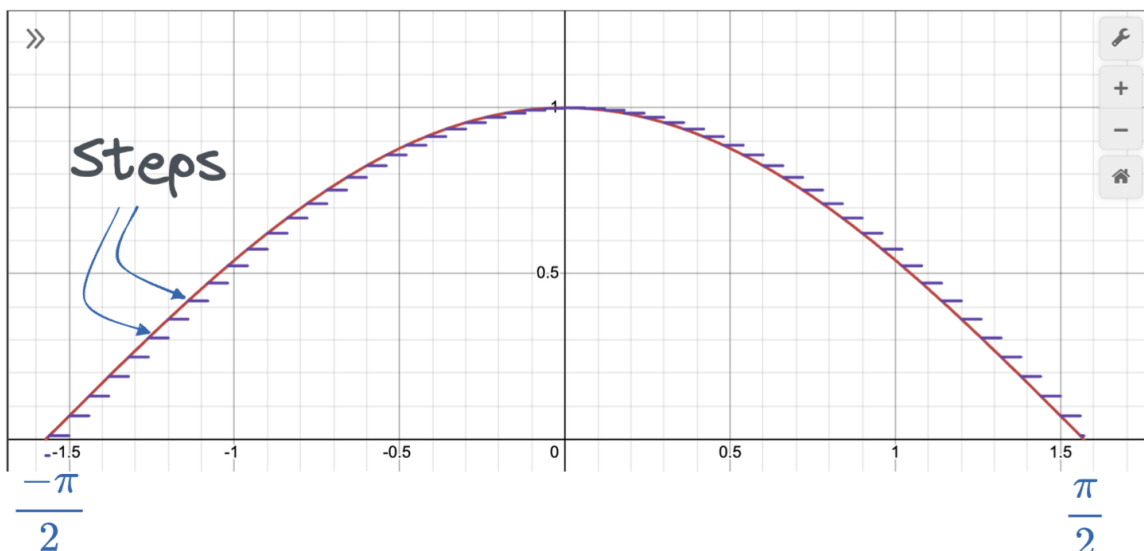
## ۱. فصل اول

### مقدمه

در پروژه های قبل تر با شبکه عصبی MLP آشنا شدیم و دیدیم که سنگ بنای بسیاری از شبکه عصبی های پیچیده امروز بر پایه همین شبکه عصبی پایه ای چیده شده است. نحوه عملکرد این شبکه بدین صورت است که با چیدن چندین لایه پشت سرهم که هرلایه یک تبدیل خطی به عمل میاورد و گذر دادن خروجی های بدست آمده از توابع فعالساز برای کسب خاصیت غیرخطی بودن، میتوانیم یک شبکه عصبی چندلایه داشته باشیم. ایده اصلی این معماری بر اساس قضیه جهانی تخمین است، به بیان ساده میتوان با استفاده از تعداد مناسبی نورون تمامی توابع پیوسته موجود در  $\mathbb{R}^n$  را به مقدار خوبی تقریب زد.

$$|f(x) - \hat{f}(x)| < \epsilon$$

به بیان ریاضی به ازای هر تابع دلخواهی مثل  $f(x)$  که در  $\mathbb{R}^n$  قرار دارد و به ازای هر  $\epsilon$  دلخواه، یک شبکه عصبی موجود است که میتواند با خطای اپسیلون آن را تخمین بزند. همچنین میدانیم که در هر شبکه عصبی MLP از توابع فعالساز یکسان در تمام لایه ها برای ایجاد کردن خاصیت غیرخطی بودن در خروجی داده ها استفاده میکنیم که از جمله این توابع میتوان به سیگموئید و ReLU اشاره کرد که این قضیه به ازای این توابع فعالساز در دهه ۸۰ میلادی بررسی و اثبات شده است. ایده ی اساسی این قضیه در این است که در یک بازه ی متناهی میتوان هر تابع پیوسته ای را با یک تابع چند قطعه ای تقریب زد.



و این بدین معنی است که یک شبکه عصبی با یک لایه مخفی میتواند هر نورون خود را به یکی از این قطعات نگاشت کند و با یکسری وزن و یک بایاس میتواند محدوده ای که این نورون پوشش میدهد را مشخص کرد.

حال اگر ورودی های دریافت شده در ناحیه مشخص شده توسط نورون قرار بگیرند وزن مثبت و بزرگی به آنها نسبت داده میشود که توسط توابع فعالساز مثل سیگموئید به یک نزدیکتر میشوند و ورودی هایی که از ناحیه مشخص شده توسط نورون مدنظر فاصله دارند وزن های منفی تری دریافت کرده و توسط تابع فعالساز مثل سیگموئید به ۰ نزدیکتر میشوند.

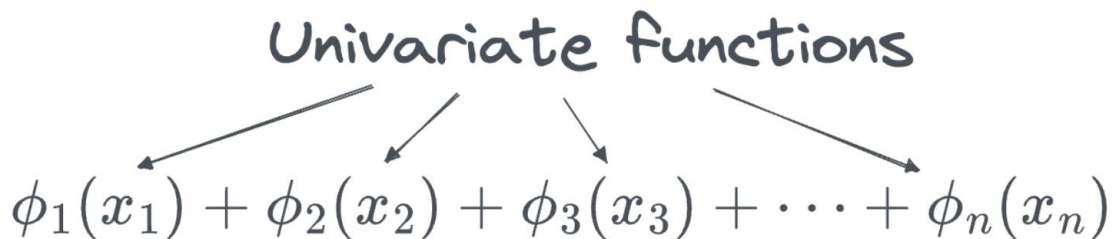
## Kolmogorov-Arnold Representation Theorem

این قضیه که یک قضیه دیگر در زمینه تخمین و تقریب توابع است بیان میکند که یک تابع چند متغیره و پیوسته را میتوان با تعداد متناهی تابع تک متغیره پیوسته تخمین زد.

$$y = F(x_1, x_2, x_3, \dots, x_n)$$

تابع بالا را درنظر بگیرید، میتوان این تابع را به شکل زیر نوشت:

Univariate functions



$$\phi_1(x_1) + \phi_2(x_2) + \phi_3(x_3) + \dots + \phi_n(x_n)$$

که در نهایت میتوان تابع  $F$  را به صورت زیر نوشت:

$$\begin{array}{cc}
 \text{Multivariate} & \text{Composition of} \\
 \text{continuous} & \text{univariate functions} \\
 \text{function} & 
 \end{array}$$

$$F(x_1, x_2, x_3, \dots, x_n) = \sum_{j=1}^m \psi_j \left( \sum_{i=1}^n \phi_{ij}(x_i) \right)$$

حال اگر عبارت بالا را باز کنیم به فرمت زیر میرسیم:

$$\begin{array}{cc}
 \text{Multivariate continuous} & \text{Composition of} \\
 \text{function} & \text{univariate functions}
 \end{array}$$

$$\begin{aligned}
 F(x_1, x_2, x_3, \dots, x_n) = & \psi_1(\phi_{11}(x_1) + \phi_{21}(x_2) + \dots + \phi_{n1}(x_n)) \\
 & + \psi_2(\phi_{12}(x_1) + \phi_{22}(x_2) + \dots + \phi_{n2}(x_n)) \\
 & \vdots \\
 & + \psi_m(\phi_{1m}(x_1) + \phi_{2m}(x_2) + \dots + \phi_{nm}(x_n))
 \end{aligned}$$

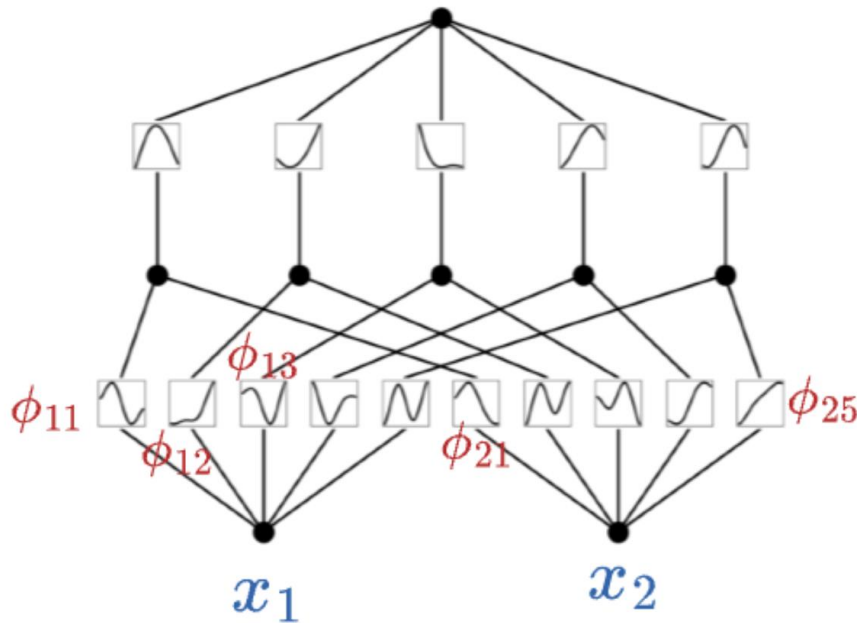
همانطور که میبینیم توانستیم یک تابع پیوسته و چندمتغیره مثل  $F$  را به صورت چندین تابع تک متغیره و پیوسته بنویسیم.

برای مثال میتوانیم  $F(x, y) = xy$  به شکل زیر بازنویسی کنیم:

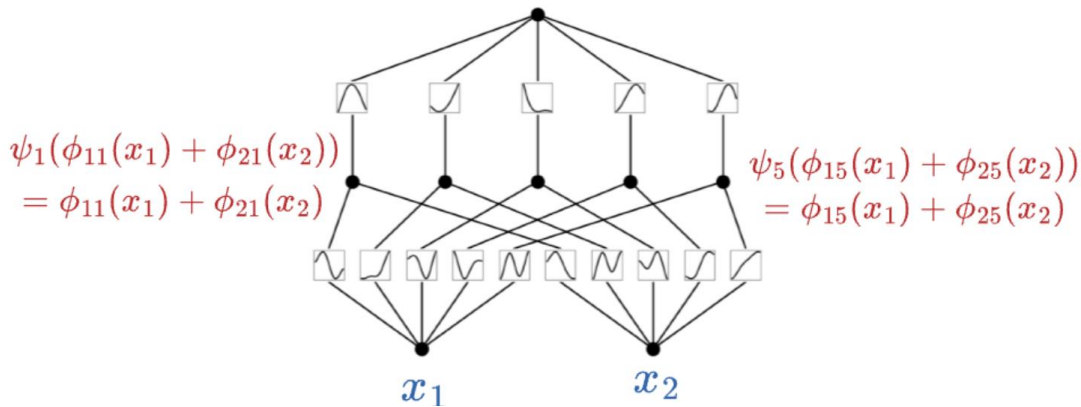
$$\begin{array}{cc}
 \text{Multivariate continuous} & \text{Composition of} \\
 \text{function} & \text{univariate functions}
 \end{array}$$

$$\begin{array}{ccc}
 F(x, y) = xy & F(x, y) = \boxed{\exp}(\boxed{\log(x)} + \boxed{\log(y)}) & \\
 \psi & \phi_x & \phi_y
 \end{array}$$

همین قضیه سنگ بنای ایده ی شبکه عصبی KAN است، از همین الان بنظر میرسد که این نحوه نمایش باید دقت بیشتری را به ازای تعداد مساوی پارامتر نسبت به MLP خروجی دهد چرا که در اینجا دیگر صحبت از تقریب و خطا نیست، عملاً داریم نحوه نمایش را تغییر میدهیم.



شکل بالا شماتیک یک شبکه عصبی KAN را نمایش میدهد، همانطور که میبینیم هر یال در حال تخمین یک تابع یک متغیره به ازای یک ورودی مثل  $x_1$  میباشد و در نهایت ان توابع که روی یال های شبکه قرار دارند با یکدیگر جمع شده و وارد لایه بعدی میشوند.

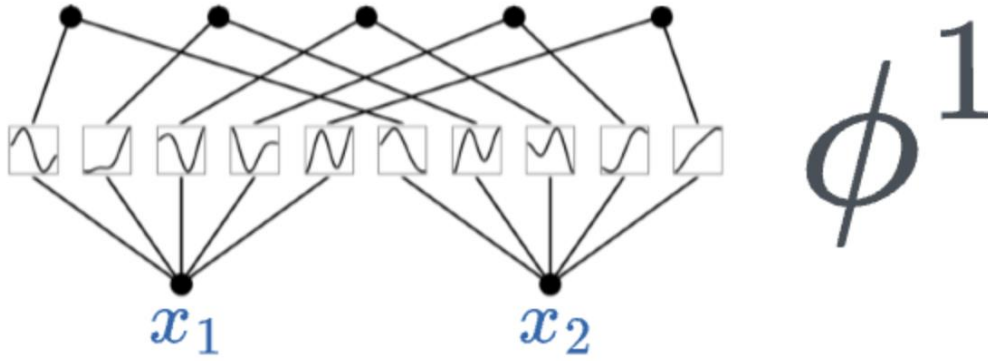


و به همین راحتی توانستیم یک لایه در KAN را تشکیل بدهیم.

تا همینجا تفاوت دیگری که این نوع معماری نسبت به MLP دارد، تفسیرپذیری آن است. در معماری قبلی، تعداد زیادی نورون داریم که در هریک وزن و بایاس قرار گرفته است و در حال انجام یک تبدیل خطی هستند همچنین به دلیل اینکه از قضیه اساسی تخمین استفاده میکنیم تفسیرپذیری هر لایه و نورون و اینکه چه ویژگی هایی را از داده استخراج میکنند کار مشکلی است ولی در شبکه عصبی جدید به دلیل اینکه داریم از نمایش دقیق تابع استفاده میکنیم و کاملاً گام به گام آن را بر اساس توابع پیوسته و تک متغیره میسازیم احتمال اینکه تفسیرپذیر تر باشند بیشتر است.

آموزش شبکه KAN:

ابتدا نحوه نمایش تبدیل های هر لایه را با یکدیگر قرار داد میکنیم.



$\phi^1$  denotes the transformation in the first layer

کاری که میکنیم این است که با داده های  $(x_1, x_2, \dots, x_n)$  را دریافت کرده و تبدیل  $\phi^1$  روی آنها انجام میدهیم، حال برای اینکه نگهداری این تبدیل آسان تر باشد باید از فرم ماتریسی آن استفاده کنیم. به همین دلیل ماتریس تبدیل هر لایه را که در پیاده سازی آن را grid مینامیم به شکل زیر نمایش داده میشود.

$$\phi^1 = \begin{bmatrix} \phi_{11} & \phi_{12} & \dots & \phi_{1n} \\ \phi_{21} & \phi_{12} & \dots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1} & \phi_{m2} & \dots & \phi_{mn} \end{bmatrix}$$

که در اینجا  $n$  سائز ورودی لایه،  $m$  سائز خروجی شبکه و هر  $\phi$  یک تابع تک متغیره مختص به ورودی  $i$ -ام است که جلوتر درمورد نحوه نمایش آن صحبت میکنیم. حال برای عمل فوروآدینگ و تغذیه کردن شبکه کافی است که بردار ورودی را دریافت کرده و آن را در ماتریس  $\phi$  ضرب کنیم و بدین صورت ورودی لایه بعدی را بدست می آوریم.



با نگهداری توابع تک متغیره  $\phi$  در قالب یک ماتریس، خیلی راحت میتوانیم توابع تک متغیره خود را در این ورودی ها اثر داده و خروجی های حاصل شده را به فرمتی که لایه بعدی از ما انتظار دارد با یکدیگر جمع کرده و آنها را تحویل دهیم.

$$z^1 = \begin{bmatrix} \text{Function applied to } x_1: \boxed{\phi_{11}(x_1)} + \phi_{12}(x_2) + \dots + \boxed{\phi_{1n}(x_n)} \\ \phi_{21}(x_1) + \phi_{22}(x_2) + \dots + \phi_{2n}(x_n) \\ \vdots \\ \phi_{m1}(x_1) + \phi_{m2}(x_2) + \dots + \phi_{mn}(x_n) \end{bmatrix}$$

که در اینجا  $z^1$  خروجی لایه اول است، پس کل شبکه عصبی KAN را میتوانیم به فرمت زیر بنویسیم:

$$KAN(x) = \phi^L ( \phi^{L-1} ( \dots ( \phi^2 ( \phi^1(x) ) ) ) ) )$$

که در اینجا  $x$  ورودی ما بوده و  $\phi$  ماتریس تبدیل هر لایه بوده و  $L$  تعداد لایه ها را نمایش میدهد. درحالی که با نگاه کردن به فرمول خروجی MLP که به فرمت زیر است:

$$NN(x) = \theta^L ( \sigma(\theta^{L-1} ( \dots ( \sigma(\theta^2 ( \sigma(\theta^1(x)) ) ) ) ) ) ) )$$

میتوانیم چندین تفاوت را دریابیم.

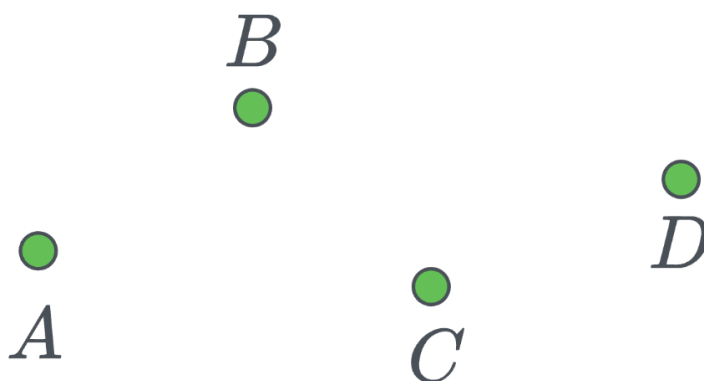
اول اینکه هر  $\theta_i$  یک تبدیل خطی را نمایش میدهد که برای دریافت خاصیت غیرخطی بودن از یک تابع فعالساز  $\sigma$  عبور میکند که در کل شبکه یکسان است.

درحالی که در KAN هر  $\phi$  یک ماتریس تبدیل است که درایه های آن همگی یکسری توابع غیرخطی تک متغیره و پیوسته بوده که عملاً میتوانند با تعداد کمتری پارامتر تخمین بهتری به عمل بیاورند.

حال این سوال مطرح میشود که این توابع غیرخطی و مشتق پذیری که در شبکه عصبی KAN بکار میرود چگونه بدست می آیند، برای این منظور ابتدا باید نگاهی به Bezier curve بیندازیم.

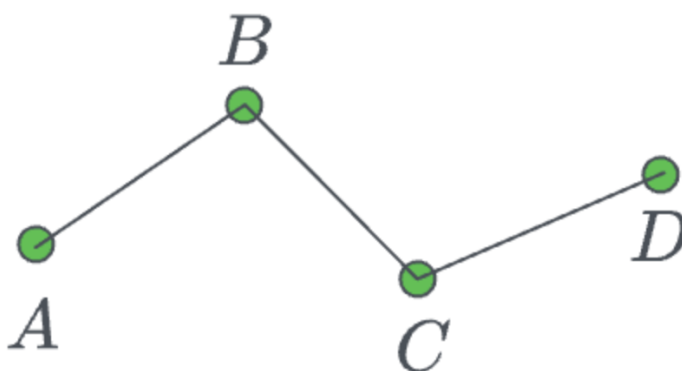
فرض کنید که یک کرکتر در بازی میخواهد به گونه ای حرکت کند که از ۴ نقطه زیر عبور کند:

Character



واضح ترین راه این است که یک خط مستقیم بین هر دو نقطه در نظر بگیریم که مسیری به شکل زیر میسازد:

Character



بدیهی است که این نوع حرکت اصلا ملایم نیست و ما نیازمند به داشتن مسیری ملایم تر بین این ۴ نقطه هستیم.

یک راه این است که مثلا از یک منحنی درجه ۳ با فرمت کلی زیر در نظر بگیریم:

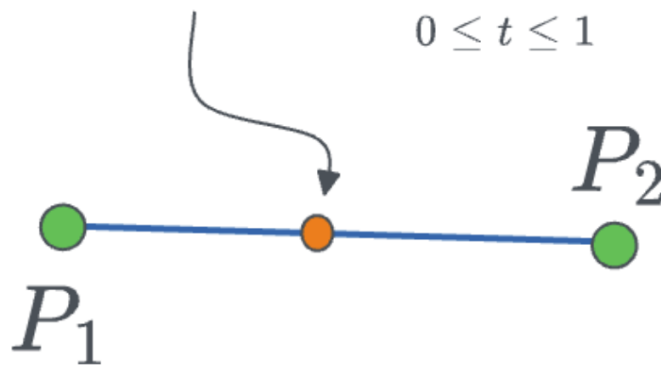
$$f(x) = ax^3 + bx^2 + cx + d$$

و سعی کنیم با جایگذاری این ۴ نقطه پارامترهای مربوطه را بدست آوریم که از نظر محاسباتی اصلاً مقرون به صرفه نیست.  
راه حل این مشکل Bezier Curves هستند.

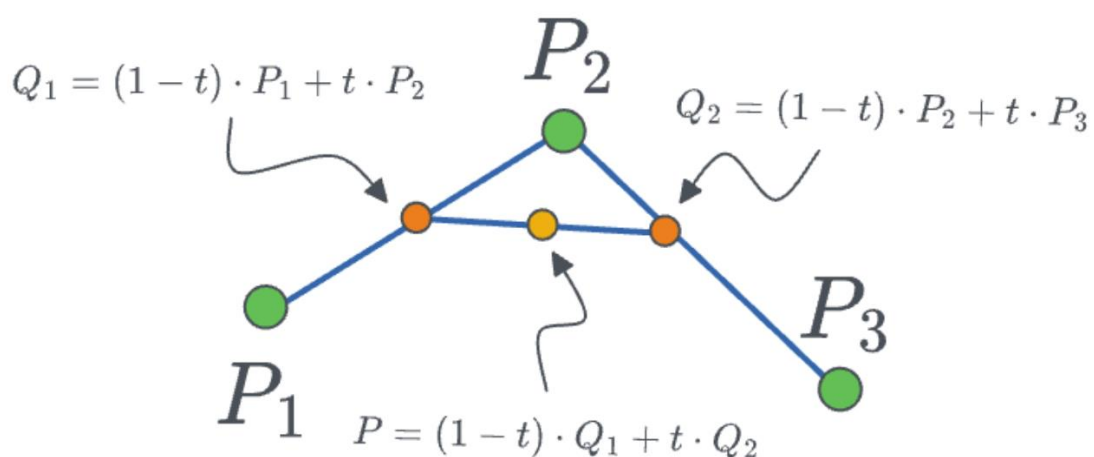
$$P = (1 - t) \cdot P_1 + t \cdot P_2$$

$$0 \leq t \leq 1$$

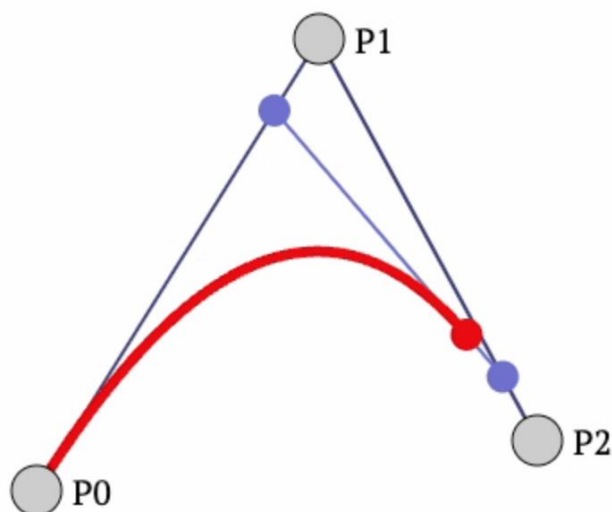
Character



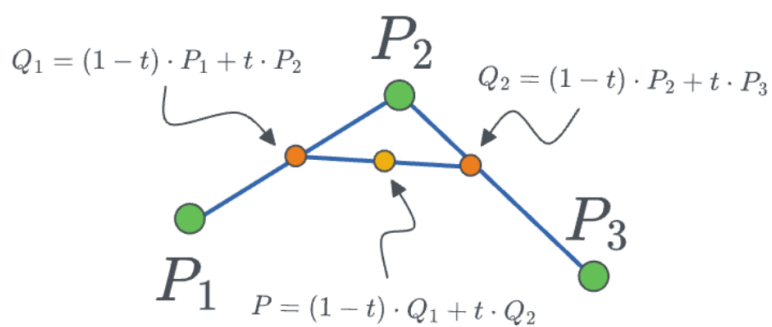
در اصل این منحنی‌ها یک ترکیب خطی محدب از نقاطی است که در تصویر قرار داشته و ما می‌خواهیم از آنها عبور کنیم، با حرکت دادن  $t$  از ۰ تا ۱، می‌توانیم یک میانگین ملایمی از میانگین‌های بدست آمده از نقاط دیگر بدست آوریم که منجر به تشکیل یک مسیر مناسب و مطلوب برای ما میشوند.  
برای مثال برای تشکیل یک منحنی بزیر در سه نقطه به شکل زیر خواهیم رسید:



که با محاسبه تخمین در نظر گرفته شده به منحنی نهایی زیر میرسیم:



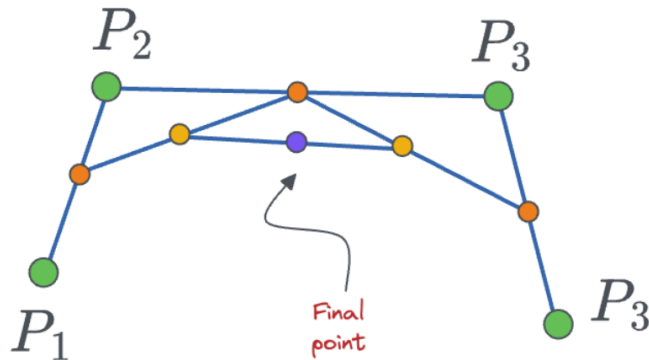
با ساده کردن معادلات بالا به معادله ای به فرمت زیر می‌رسیم:



$$B(t) = (1-t)^2 P_1 + 2(1-t)t P_2 + t^2 P_3, t \in [0, 1]$$

To obtain this path, substitute all values in terms of  $P_1$ ,  $P_2$ , and  $P_3$ .

حال اگر ۴ نقطه در نظر بگیریم نیز به فرمت زیر می‌رسیم:



$$B(t) = (1-t)^3 P_1 + 3(1-t)^2 t P_2 + 3(1-t) t^2 P_3 + t^3 P_4, \quad t \in [0, 1]$$

To obtain this path, substitute all values in terms of  $P_1$ ,  $P_2$ ,  $P_3$  and  $P_4$ .

با مشاهده و مقایسه فرمول های بالا به فرم کلی زیر میرسیم:

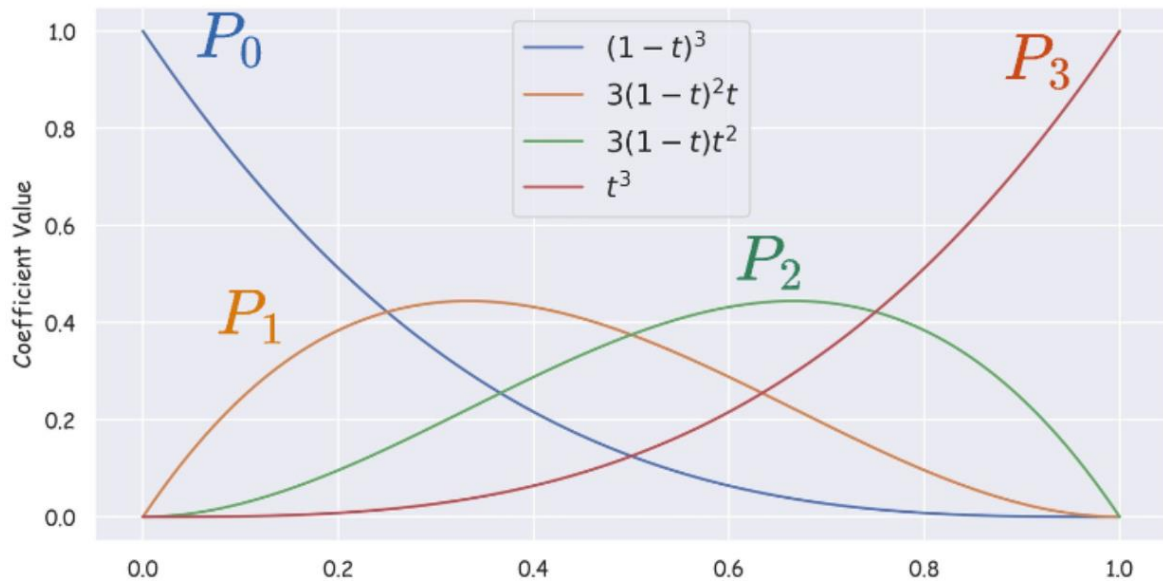
$$B(t) = \sum_{i=0}^n \left( \binom{n}{i} (1-t)^{n-i} t^i \right) P_i ; \quad \binom{n}{i} = \frac{n!}{i!(n-i)!}$$

Binomial coefficient

$$= \sum_{i=0}^n c_{i,n}(t) P_i$$

که هر  $c_{i,n}$  یک تابع بر حسب  $t$  بوده که در نقطه ی  $P_i$  اثر داده شده و تاثیر تابع  $c_{i,n}$  را در شکل گیری منحنی نهایی را مشخص میکند.

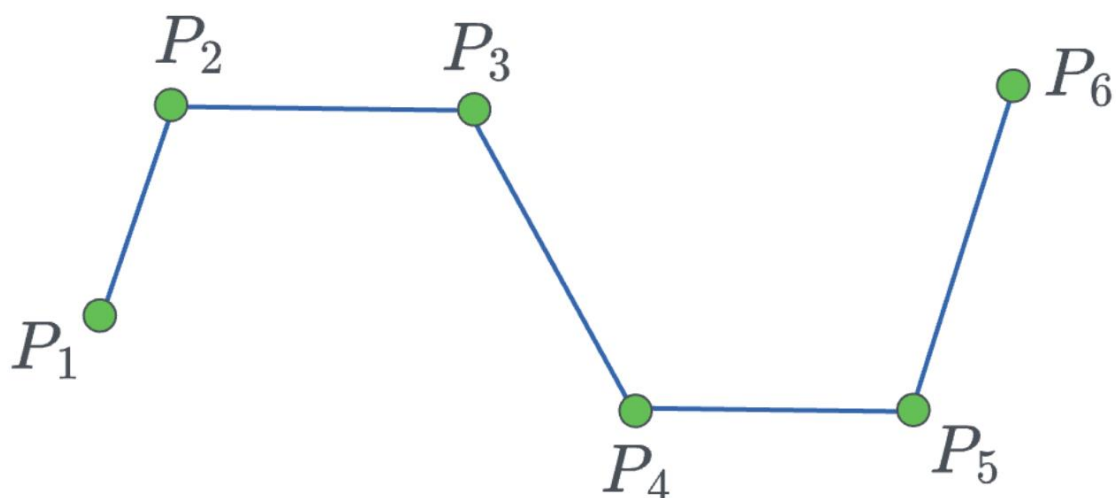
در اصل هر  $c_{i,n}$  یک base function است که  $P_i$  نقش یک control point/coefficient را دارد. برای مثال در نمودار زیر میتوانیم میزان اثرگذاری هر coefficient در شکل گیری منحنی نهایی با حرکت  $t$  از ۰ تا ۱ را تشخیص بدهیم.



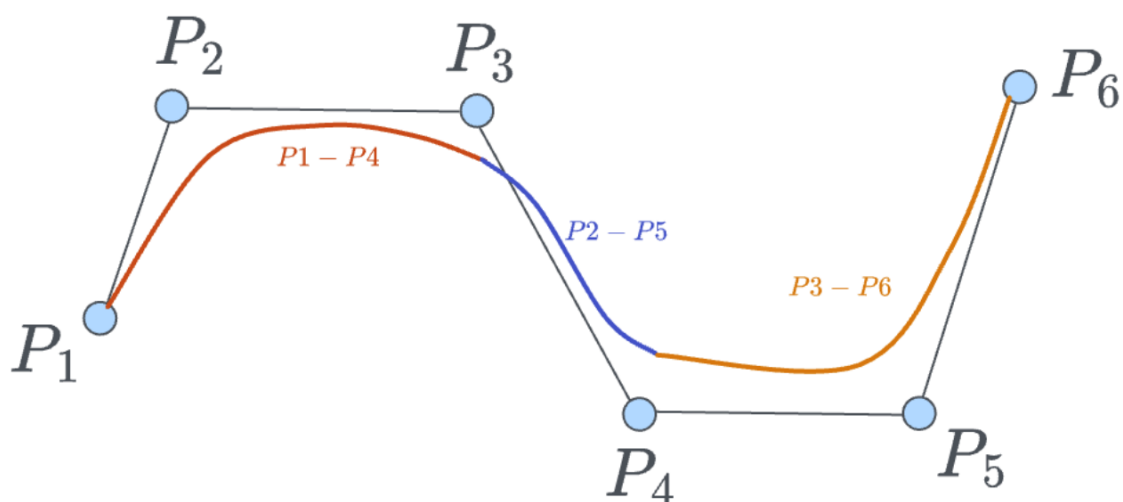
برای مثال در نمودار بالا در نقطه ی  $t = 0$  همه ی coefficient ها به جز  $P_0$  مقدار صفر دارند که نشان میدهد نمودار از نقطه  $p_0$  شروع میشود. با بیشتر شدن  $t$  میبینیم که منحنی به هریک از coefficient های  $p_2, p_3, p_4$  به ترتیب نزدیک میشود. حال همه چیز درست بنظر میرسد ولی ایا منحنی های بزیر راه حل مشکل ما هستند؟ پاسخ منفی است، اگر به فرمول بدست آوردن base function ها نگاه کنید، میبینیم که مرتبه زمانی این منحنی بسیار زیاد بوده و برای مثلا ۱۰۰ نقطه اصلا مناسب نیست. راه حل مناسب B spline است.

### B Spline:

بی اسپلاین یک راه مناسب و بهینه تر برای تشکیل منحنی ها مخصوصا برای نقاط زیاد ارائه میدهند. برخلاف منحنی های درجه بالا، بی اسپلاین از دنباله ای از منحنی های درجه پایین برای تشکیل منحنی خود استفاده میکند. به عبارتی دیگر به جای اینکه یک منحنی بزیر درجه بالا برای عبور از تمامی نقاط درنظر بگیریم، از چندین منحنی بزیر درجه پایین استفاده کرده و آنها را به یکدیگر متصل میکنیم. برای مثال شکل زیر را درنظر بگیرید:



تشکیل بی اسپالین روی این نقاط نتیجه ای به شکل زیر میدهد:



در شکل بالا ۶ کنترل پوینت داریم و در حال رسم منحنی های بزیر درجه ۳ هستیم، در نتیجه ۳ منحنی باید رسم کنیم ( $n-k$  منحنی در حالت کلی)  
 حال برای اینکه یک بی اسپالین رفتار ملایم و پیوسته ای داشته باشد باید شرط پیوستگی کلاس های  $C0, C1, C2$  را داشته باشد. برای مثال پیوستگی از کلاس  $C2$  را توضیح میدهیم:  
 مقدار مشتق در نقاط مرزی منحنی های بزیر باید برابر باشد.

حال میتوانیم فرمول بی اسپلاین با درجه  $k$  را به شکل تعریف کنیم:

$$S(t) = \sum_{i=0}^n N_{i,k}(t) P_i,$$

که در آن  $P_i$  ها کنترل پوینت ها بوده و  $N_{i,k}$  نیز base function هایی هستند که آنها را میتوان با روش بازگشتی cox-de boor محاسبه کرد،  $t$  نیز مقدار ورودی به منحنی بی اسپلاین بدست آمده را نشان میدهد.

فرمول بازگشتی cox-de boor نیز به فرم زیر است:

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

$$N_{i,j}(t) = \frac{t - t_i}{t_{i+j} - t_i} N_{i,j-1}(t) + \frac{t_{i+j+1} - t}{t_{i+j+1} - t_{i+1}} N_{i+1,j-1}(t).$$

که در آن  $t_i$  نیز knot points هستند.

برای محاسبه ی توابع پایه هر کنترل پوینت میتوان شماتیک مثلثی مثل شماتیک زیر در نظر گرفت:

$$\begin{array}{ccccccc} N_{0,0}(t) & \rightarrow & N_{0,1}(t) & \cdots & N_{0,k-1}(t) & \rightarrow & N_{0,k}(t) \\ & \nearrow & & & & \nearrow & \\ N_{1,0}(t) & \rightarrow & N_{1,1}(t) & \cdots & N_{1,k-1}(t) & & \\ \vdots & & \vdots & & & & \\ N_{n-1,0}(t) & \rightarrow & N_{n-1,1}(t) & & & & \\ & \nearrow & & & & & \\ N_{n,0}(t) & & & & & & \end{array}$$

حال برای اینکه بتوانیم این توابع پایه را بدست آوریم نیاز به knot points داریم.

Knot points به دنباله ای مثل  $T = (t_1, t_2, \dots, t_n)$  میگویند که در اصل نقش یک محور مختصات برای ما را بازی میکنند، حال اگر فواصل این نقاط با یکدیگر برابر باشد به بی اسپلاین شکل گرفته، uniform b-spline گفته میشود.

ما در این پیاده سازی با uniform b-spline سروکار داریم.



و باید بدانیم که متغیر  $t$  که در نظر میگیریم در بازه ی knot point ها قرار داشته و مشابه با منحنی های زیر، از ۰ تا  $t_m$  حرکت کرده و در هر بازه تاثیر یک تابع پایه ای را بیشتر از باقی توابع در نظر میگیرد. طول بازه نات پوینت ها وابسته به تعداد کنترل پوینت ها و درجه بی اسپلاین است و از رابطه ی زیر بدست می آید:

$$m = k + n + 1$$

برای مثال برای ۵ کنترل پوینت و منحنی درجه ۳، ۹ نات پوینت خواهیم داشت. برای مثال اگر بخواهیم توابع پایه ای را روی ۳ کنترل پوینت و درجه ۳ تعریف کنیم به سه تابع پایه ای زیر روی ۶ نات پوینت میرسیم که از همان روش بازگشتی اشاره شده بدست آمده اند.

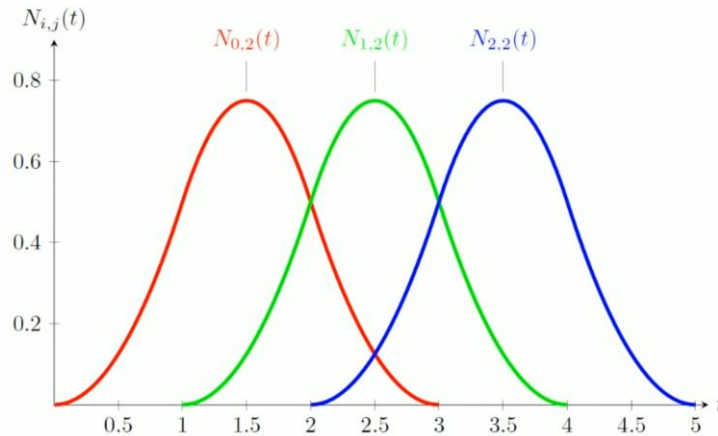
$$N_{0,2}(t) = \begin{cases} \frac{1}{2}t^2 & \text{if } 0 \leq t < 1 \\ \frac{1}{2}[-2(t-1)^2 + 2(t-1) + 2] & \text{if } 1 \leq t < 2 \\ \frac{1}{2}[(t-2)^2 - 2(t-2) + 1] & \text{if } 2 \leq t < 3 \\ 0 & \text{otherwise} \end{cases}$$

$$N_{1,2}(t) = \begin{cases} \frac{1}{2}(t-1)^2 & \text{if } 1 \leq t < 2 \\ \frac{1}{2}[-2(t-2)^2 + 2(t-2) + 2] & \text{if } 2 \leq t < 3 \\ \frac{1}{2}[(t-3)^2 - 2(t-3) + 1] & \text{if } 3 \leq t < 4 \\ 0 & \text{otherwise} \end{cases}$$

$$N_{2,2}(t) = \begin{cases} \frac{1}{2}(t-2)^2 & \text{if } 2 \leq t < 3 \\ \frac{1}{2}[-2(t-3)^2 + 2(t-3) + 2] & \text{if } 3 \leq t < 4 \\ \frac{1}{2}[(t-4)^2 - 2(t-4) + 1] & \text{if } 4 \leq t < 5 \\ 0 & \text{otherwise} \end{cases}$$

نکته قابل توجه این است که هر سه تابع یکسان هستند و از شیفت دادن تابع پایه ای اول به مقدار  $k$  واحد بدست آمده اند.

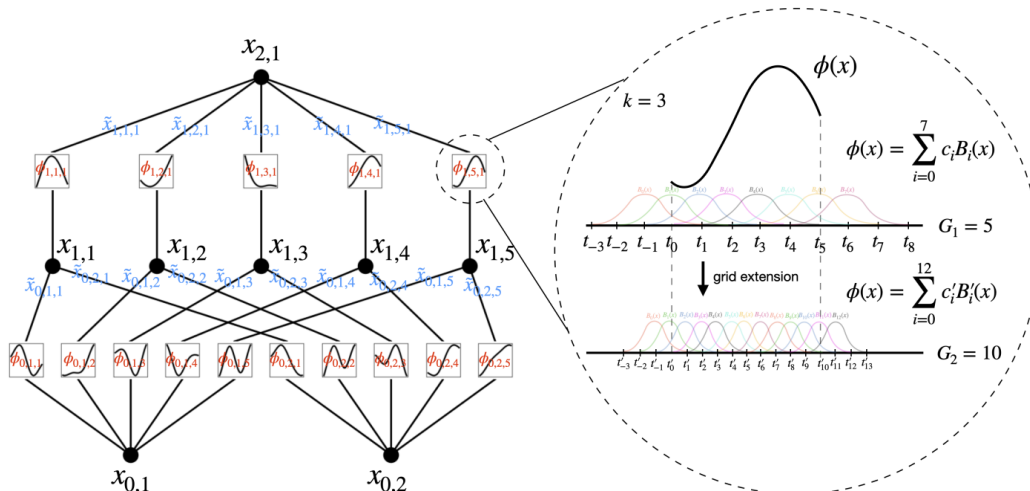
در شکل زیر میتوانیم نمودار این سه تابع پایه ای روی نات پوینت های در نظر گرفته شده را ببینیم و به ازای  $t$  های مختلف میتوانیم تاثیر هریک از این سه تابع پایه ای مختلف در منحنی نهایی را مشاهده کنیم :



البته فراموش نکنیم که کنترل پوینت ها مثل وزن های این توابع عمل کرده و ترکیب خطی این توابع و کنترل پوینت ها منجر به شکل گیری یک منحنی جدید میشود. حال دلیل اینکه چرا در شبکه عصبی KAN از بی اسپلاین استفاده میشود کمی واضح تر است، اولین و مشخص ترین دلیل بار محاسباتی کم آن نسبت به بزیرهاست و دلیل دوم این است که بی اسپلاین خاصیت کنترل محلی را فراهم میکند به عبارتی دیگر با تغییر یک کنترل پوینت کل نمودار به هم نمیریزد و هر چه از ناحیه مربوط به آن کنترل پوینت فاصله میگیریم تاثیر تغییر آن کمتر و کمتر میشود که این خاصیت منجر به داشتن حافظه طولانی تری در شبکه عصبی ما میشود.

پس به عنوان جمع بندی میتوان اینگونه در نظر گرفت که نات پوینت ها مثل یک محور مختصات برای ما عمل کرده که روی آن میتوانیم توابع پایه ای را براساس تعداد کنترل پوینت ها و درجه مدنظر بدست آورده و کنترل پوینت هارا مثل وزن هایی در نظر گرفته که نشان میدهد هر تابع پایه ای به چه میزان در شکل گیری منحنی نهایی در ناحیه مختص به خود نقش دارد. در شبکه عصبی ما همین کنترل پوینت ها هستند که پارامترهای قبل آموزش هستند و تغییر دادن آنها میتواند منجر به شکل گیری یک اسپلاین جدید بشود و گرنه که توابع پایه ای به ازای درجه و تعداد کنترل پوینت ها، مشخص و معلوم هستند.

در شکل زیر میتوانیم جمع‌بندی مطالبی که تا الان بیان شده است را یکجا ببینیم:



همانطور که در شماتیک سمت راست میبینیم، میخواهیم اسپلاین های ورودی مثل  $x$  را به ازای درجه ۳ و تعداد ۵ نات پوینت و ۸ کنترل پوینت را در نمودار بالا ببینیم. که هر  $c_i$  در نقش یک وزن برای تابع پایه ای مربوط به خود عمل میکند.

پس به طور کلی مراحل کلی یادگیری و خروجی دادن شبکه عصبی ما به شکل زیر است:

- همانند وزن های MLP، مقادیر کنترل پوینت را با استفاده از توزیع های خاص و مناسب مثل توزیع  $Xavier, c_i \sim N(0, \sigma^2)$  مقدار دهی کنید که در اینجا سیگما مقداری کوچک مثل ۰.۱ دارد.
- میتوانید وزن هارا براساس توزیع Xavier وزن دهی کنید.
- با استفاده از ماتریس های تبدیل، عملیات feed forward را انجام دهید.
- در مرحله یادگیری، خطا را محاسبه کرده و با استفاده از optimizer این کنترل پوینت هارا اپدیت کنید.

در مقاله نیز اشاره شده که برای کنترل بهتر روی خروجی spline و خروجی کلی یک تابع از وزن هایی مثل  $w_s, w_b$  استفاده شده است که میتوان برای راحتی آنها را در نظر نگرفت. همچنین از یک تابع پایه مثل Silu استفاده کرده و آن را با مقدار محاسبه شده جمع میکنیم.

در زیر میتوانید نحوه وزن دهی اولیه وزن های  $w$  را براساس توزیع های نرمال و xavier مشاهده کنید:

If sampling for uniform distribution

$$w \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right)$$

input dimension      output dimension

If sampling for normal distribution

$$w \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

input dimension      output dimension

در پاسخ به اینکه نقش  $W$  در محاسبه ی خروجی اسپلین چیست باید گفت که در مقاله اشاره شده که دو وزن  $w_b$ ,  $w_s$  در نظر گرفته شده اند که نقش کنترل کننده در میزان تاثیر اسپلین محاسبه شده و تابع پایه ای ما که با  $b(x)$  نمایش داده میشود را دارند، که تابع پایه ای و ثابت ما در اینجا  $\text{Silu}(x)$  است.

اما برای راحتی میتوان آنها را در نظر نگرفت.

پس به طور کلی هر grid را میتوان به فرمت زیر نمایش داد:

$$w(b(x) + \sum_i c_i B_i(x))$$

$$\phi^1 = \begin{bmatrix} \phi_{11}(\cdot) & \phi_{12}(\cdot) & \dots & \phi_{1n}(\cdot) \\ \phi_{21}(\cdot) & \phi_{22}(\cdot) & \dots & \phi_{2n}(\cdot) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{m1}(\cdot) & \phi_{m2}(\cdot) & \dots & \phi_{mn}(\cdot) \end{bmatrix}$$

و برای تغذیه کردن شبکه کافی است لایه هارا پی در پی در یکدیگر اثر دهیم:

$$KAN(x) = \phi^L ( \phi^{L-1} ( \dots ( \phi^2 ( \phi^1(x) ) ) ) ) )$$

حال با درنظر گرفتن توضیحات بالا، پیاده سازی پروژه چیزی جز ترجمه موارد ذکر شده به زبان کد نیست.

## ۲. فصل دوم

### پیاده سازی KAN با استفاده از Pytorch

پیاده سازی انجام شده در سه بخش در نظر گرفته شده است،

Layer Class-

KAN Class-

Train loop and Test-

از بخش Layer Class شروع کرده و به ترتیب به سراغ باقی بخش ها نیز میرویم:

Layer Class:

این کلاس از توابع زیر تشکیل شده است:

`__init__()`:

بدیهی است که این تابع کانستراکتور کلاس لایه است و پارامتر هایی که برای تشکیل یک لایه به آن نیازمند هستیم را در این تابع مشخص میکنیم، پارامتر هایی که در این تابع به عنوان ورودی دریافت میشوند شامل `in_feature` که سایز فضای فیچر های ورودی، `out_feature` که سایز فیچر های خروجی از لایه، `grid_range` که بیان میدارد نات پوینت ها در چه بازه ای باید قرار داشته باشند، `grid_size` که تعداد نات پوینت هارا بیان میدارد، `spline_order` که درجه بی اسپلاین را نشان میدهد و مقدار دیفالت آن ۳ است و `sigma` که پارامتری برای استاندارد سازی است و در footnote ۲ در صفحه ۶ توصیف شده است.

در اصل `grid` در پیاده سازی ما نقش transformation matrix را بازی میکند که هر درایه آن از یک بی اسپلاین تشکیل شده است، در اصل گرید یک ماتریس سه بعدی است که ورودی ها را گرفته و آنها را به فضای لایه بعدی نگاشت میکند و در هر درایه خود تعدادی نات پوینت نگهداری میکند که با آنها میتوانیم توابع پایه ای را به ازای یک  $x$  دریافتی به روش بازگشتی cox de-boor محاسبه کرده و در نهایت با تاثیر دادن کنترل پوینت ها بتوانیم اسپلاین هارا بوجود بیاوریم.

پس از مشخص شدن این پارامتر ها نوبت به تشکیل گرید و یکسری پارامتر دیگر با استفاده از همین پارامتر های دریافتی میرسد.

ابتدا `grid_spacing` را محاسبه میکنیم که فاصله بین دو نات پوینت متوالی را محاسبه میکند و در نهایت با استفاده از `grid_spacing` و `spline order` و `grid_size` اکنون میتوانیم گرید را تشکیل دهیم.

تشکیل دادن `w_b` مشابه با مقاله است که نقش پارامتر کنترلی `base function` را بازی میکند که قرارداد آن در فرایند فورواردینگ اختیاری است.

مقدار اولیه آن از روش `xavier` بدست می آید که قبل تر به آن اشاره شده است.

`Base function` نیز تابع `silu` در نظر گرفته شده است که در مقاله اشاره شده همانند `residual connections` ها رفتار میکند.

در نهایت با استفاده از تابع `reset_parameters()` `coeff` ها را که با `c_i` نمایش میدهم، مقداردهی میکنیم.

```
class Layer(torch.nn.Module):
    def __init__(
        self,
        in_features = 2,
        out_features = 5,
        grid_range = [-1,1],
        grid_size = 5, # Number of knot/control point in the interval -1 , 1
        spline_order = 3, # polynomial of order 2
        sigma = 0.1, # sigma in footnote 2 of page 6
    ):
        base_activation = torch.nn.SiLU(),

        super(Layer, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.grid_size = grid_size
        self.spline_order = spline_order

        grid_spacing = (grid_range[1] - grid_range[0]) / grid_size
        grid = ((torch.arange(-spline_order, grid_size + spline_order + 1) *
        grid_spacing + grid_range[0]).expand(in_features, -1))

        self.grid = grid

        self.w_b = torch.nn.Parameter(torch.Tensor(out_features, in_features)) #w_b in
        2.10 in the paper

        torch.nn.init.kaiming_uniform_(self.w_b, a=math.sqrt(5)) # Xavier
        self.c_i = torch.nn.Parameter(torch.Tensor(out_features, in_features, grid_size
        + spline_order)) #C_is in 2.12 in the paper

        self.sigma = sigma
        self.base_activation = base_activation()

        self.reset_parameters()
```



`Reset_parameters()`

همانطور که گفتیم در بخش قبلی از این تابع برای مقداردهی اولیه وزن ها یا همان coefficient ها استفاده میکنیم.

در اصل وظیفه این تابع ریست کردن تمامی پارامترهایی است که در تشکیل اسپلاین نقش دارند اعم از `w_b` و `coeff` ها.

روش مقداردهی `w_b` که مشابه قبل از روش xavier بدست می آید و برای مقدار دهی `c_i` ها با تشکیل یک نویز مناسب که ابعاد آن با گرید سازگار بوده و مقدار آن با استفاده از `sigma` استاندارد شده است و پاس دادن آن به عنوان یک آرگومان به تابع `curve2coeff`، میتوانیم مقادیر رندوم و مناسبی که مقاله به آن تاکید کرده است برای `c_i` ها بدست آوریم.

```
def reset_parameters(self):
    torch.nn.init.kaiming_uniform_(self.w_b, a=math.sqrt(5))

    noise = ((torch.rand(self.grid_size + 1, self.in_features, self.out_features)
              1 / 2) * self.sigma / self.grid_size)

    self.c_i.data.copy_(self.curve2coeff(self.grid.T[self.spline_order : -self.
        spline_order], noise,))
```

`B_spline(x)`

این تابع یک مقدار مثل `x` را دریافت میکند که ورودی لایه ی مدنظر است و درنهایت با داشتن گرید میتوانیم توابع پایه ای مناسب برای این ورودی را به روش بازگشتی cox de-boor محاسبه و مقدار دهی کنیم و به عنوان پایه های جدید بازگردانیم.

که این روش بازگشتی قبل تر توضیح داده شده است.

```

def b_splines(self, x):

    grid = self.grid
    x = x.unsqueeze(-1)

    # Initialize the bases tensor with zeros
    bases = torch.zeros((x.size(0), grid.size(0), grid.size(1) - 1), dtype=x.dtype,
                        device=x.device)

    # Calculate bases for k=0
    for n in range(x.size(0)):
        for i in range(grid.size(0)):
            for j in range(grid.size(1) - 1):
                if x[n, 0] ≥ grid[i, j] and x[n, 0] < grid[i, j + 1]:
                    bases[n, i, j] = 1.0
    # Iterate over the spline order
    for k in range(1, self.spline_order + 1):
        new_bases = torch.zeros((x.size(0), grid.size(0), grid.size(1) - k - 1),
                                dtype=x.dtype, device=x.device)

        for n in range(x.size(0)):
            for i in range(grid.size(0)):
                for j in range(grid.size(1) - k - 1):
                    left_term = 0.0
                    right_term = 0.0

                    if grid[i, j] ≠ grid[i, j + k]:
                        left_term = ((x[n, 0] - grid[i, j]) / (grid[i, j + k] - grid[i, j])) * bases[n, i, j]

                    if grid[i, j + k + 1] ≠ grid[i, j + 1]:
                        right_term = ((grid[i, j + k + 1] - x[n, 0]) / (grid[i, j + k + 1] - grid[i, j + 1])) * bases[n, i, j + 1]

                    new_bases[n, i, j] = left_term + right_term

        bases = new_bases
    # print(bases)

```

Curve2coeff(x, y)

این تابع که در بخش `reset_parameters()` نیز بکار رفته است، با دریافت یک  $x, y$ ، که همان `noise` بوده و  $x$  مقدار ورودی مورد نظر که در اینجا `grid` است (که ابعاد آن تغییر کرده است و در نهایت سائز آن `in_feature * grid_size` خواهد بود)، اقدام به حل یک دستگاه معادلاتی میکند که منجر به مشخص شدن  $c_i$  ها به عنوان وزن هایی برای توابع پایه ای موجود در هر درایه از گرید میشود.

روش حل دستگاه معادلاتی از روش `least squares` بوده که توضیح آن خارج از حوصله این گزارش کار است.

در نهایت با مرتب کردن ابعاد کنترل پوینت ها، آن ها را خروجی میدهم.

(ابعاد خروجی های `b_spline`, `y`, `c_i` ها در کامنت گذاری ها شفاف سازی شده اند.

```
def curve2coeff(self, x, y):
    # Compute the B-spline bases for the input tensor x
    spline_bases = self.b_splines(x) # (batch_size, in_features, grid_size +
    spline_order)
    spline_bases_t = spline_bases.transpose(0, 1) # (in_features, batch_size,
    grid_size + spline_order)

    # Transpose the target tensor y
    y_t = y.transpose(0, 1) # (in_features, batch_size, out_features)

    # Solve the least squares problem to find the spline coefficients
    coefficients = torch.linalg.lstsq(spline_bases_t, y_t).solution #
    (in_features, grid_size + spline_order, out_features)

    # Permute the dimensions to get the correct shape
    coeffs_permuted = coefficients.permute(2, 0, 1) # (out_features, in_features,
    grid_size + spline_order)
    return coeffs_permuted
```

:Forward(x)

این تابع وظیفه تغذیه کردن لایه را داشته و با دریافت مقدار ورودی `x`، دو مقدار را محاسبه میکند،

`Base_output` و `spline_output` که اولی تنها تابع پایه ای `silu` را مقداردهی میکند و با اثر دادن `w_b` در نقش پارامتر کنترلی این تابع مقدار `base_output` را خروجی میدهد.

`Spline_output` نیز نیز توابع پایه ای را محاسبه و مقداردهی کرده و با اثر دادن `c_i` ها، تمامی اسپلاین های لازم را در هر درایه از گرید محاسبه میکند و با عبور از یک `Linear` مقدار نهایی را خروجی میدهد.

حاصل جمع این دو خروجی محاسبه شده مقدار نهایی خروجی را به ما میدهد که با مرتب سازی ابعاد آن میتوانیم ورودی لایه بعدی را بازگردانیم

```
def forward(self, x):
    original_shape = x.shape
    x = x.reshape(-1, self.in_features)

    base_output = F.linear(self.base_activation(x), self.w_b)
    spline_output = F.linear(
        self.b_splines(x).reshape(x.size(0), -1),
        self.c_i.reshape(self.out_features, -1),
    )
    output = base_output + spline_output

    output = output.reshape(*original_shape[:-1], self.out_features)
    return output
```

کلاس KAN:

این کلاس نیز از توابع زیر تشکیل شده است:

\_\_init\_\_:

این کانستراکتور تعداد لایه های مخفی، سائز گرید که تعدادنات پوینت هارا نمایش میدهد و درجه اسپلاین و سیگما و تابع پایه ای که نوع آن Silu درنظر گرفته شده است، به ترتیب لایه های kan مناسب را ساخته و آن هارا پشت سرهم قرار میدهد.

تنها نکته حائز اهمیت در اینجا این است که in\_feature, out\_feature بر اساس سائز لایه های پنهان ساخته میشود.

```

class KAN(torch.nn.Module):
    def __init__(
        self,
        layers_hidden,
        grid_size=5,
        spline_order=3,
        sigma=0.1,
        base_activation=torch.nn.SiLU,
        grid_range=[-1, 1],
    ):
        super(KAN, self).__init__()
        self.grid_size = grid_size
        self.spline_order = spline_order

        self.layers = torch.nn.ModuleList()
        for in_features, out_features in zip(layers_hidden, layers_hidden[1:]):
            self.layers.append(
                Layer(
                    in_features,
                    out_features,
                    grid_size=grid_size,
                    spline_order=spline_order,
                    sigma=sigma,
                    base_activation=base_activation,
                    grid_range=grid_range,))

```

.Forward()

این تابع توضیح خاصی ندارد، صرفاً مقدار ورودی را به هر لایه داده و خروجی آن را به عنوان ورودی لایه بعد در نظر میگیرد.

```

def forward(self, x: torch.Tensor):
    for layer in self.layers:
        x = layer(x)
    return x

```

تست و نتیجه:

دیتاستی که برای تست این شبکه در نظر گرفته شده است، Iris بوده که شامل ۴ فیچر و ۱۵۰ نمونه میشود که کلاس آنها ۳ گونه گیاهی از نژاد Iris میباشد.

پس از پیش پردازش این دیتاست که تنها لازم بود variety را label encode کنیم، با اعمال کردن pca و کاهش ابعاد دیتاست به دو بعد، آن را نمایش داده و پراکندگی داده هارا بررسی میکنیم، دقت کنید که این روش برای پلات کردن مناسب نیست، چرا که روش pca هدفش استخراج بیشترین اطلاعات توسط واریانس در هر کامپوننت بوده که باعث میشود شکل داده هارا به هم بزند ولی در این مورد همین نحوه نمایش برای ما کفایت میکند.

```
from sklearn.preprocessing import LabelEncoder

LabelEncoder_y = LabelEncoder()
df['variety'] = LabelEncoder_y.fit_transform(df['variety'])

import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

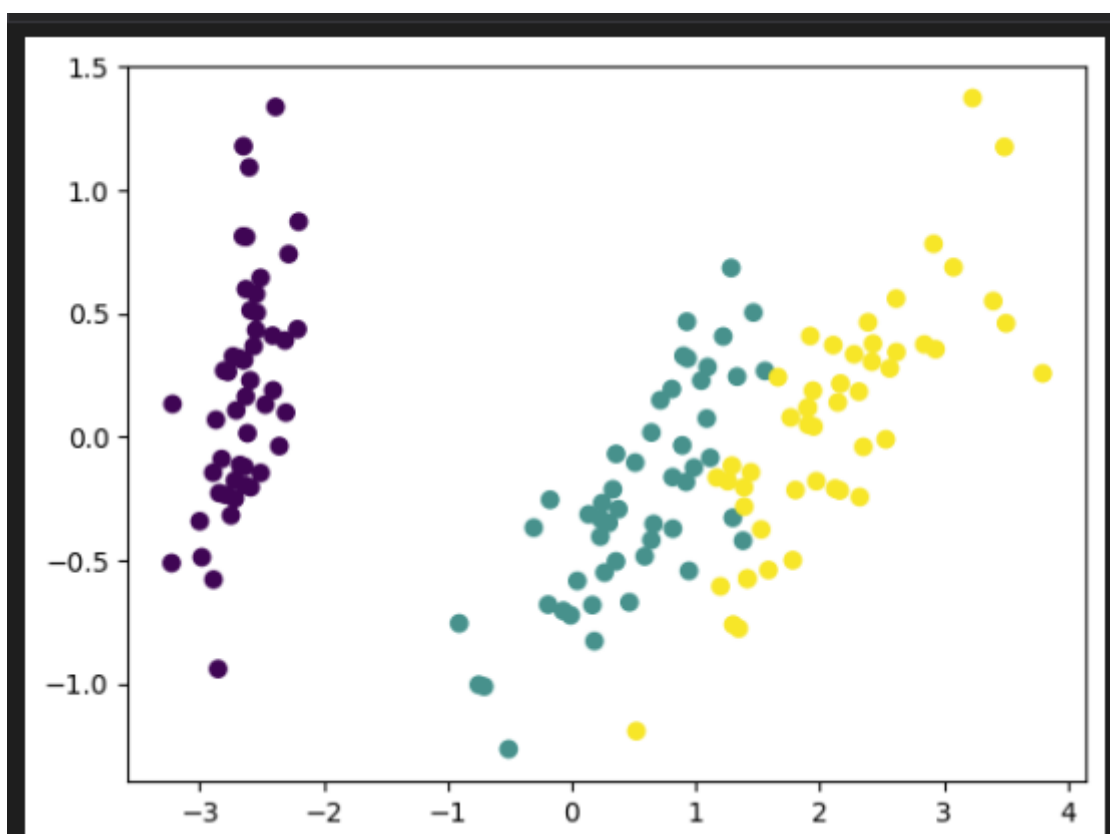
# make dataset 2D and plot it

pca = PCA(n_components=2)

X = df.drop("variety", axis=1)
y = df["variety"]

X = pca.fit_transform(X)

plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



در نهایت لازم است که اعضای دیتاست به تنسور cast شوند تا بتوانیم آنها را در شبکه تزریق کنیم.

```
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)
```

در نهایت نیز، شبکه را با ابعاد لایه پنهانی که مدنظرمان است تعریف کرده و پس از تعریف کردن یک optimizer مناسب که در اینجا adam استفاده شده است، فرایند آموزش را شروع میکنیم.

نرخ یادگیری ۰.۰۱ در نظر گرفته شده و تابع خطا نیز MSE میباشد.

```
import torch.optim as optim

model = KAN([4, 3, 1])

optimizer = optim.Adam(model.parameters(), lr = 0.01)
criterion = torch.nn.MSELoss()
```

در نهایت در حلقه یادگیری خواهیم داشت:

```
n_epochs = 20

for epoch in range(n_epochs):
    optimizer.zero_grad()
    y_pred = model(X)
    loss = criterion(y_pred, y)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch} Loss: {loss.item()}")
```

که خطوط این حلقه مشخص هستند و نیاز به توضیح اضافه تری نیست،

`Loss.backward` در هر مرحله مشتق های جزئی تمام پارامتر هارا حساب کرده و با صدا کردن `optimizer.step` روش گرادیان کاهشی بکار رفته و خروجی شبکه در مرحله بعدی بهبود میابد.



همانطور که در تصویر زیر میتوانید مشاهده کنید شبکه در حال یادگیری است:

```
Epoch 0 Loss: 1.250259518623352
Epoch 1 Loss: 0.9595373272895813
Epoch 2 Loss: 0.7835935950279236
Epoch 3 Loss: 0.7100650072097778
Epoch 4 Loss: 0.7158111929893494
Epoch 5 Loss: 0.7600979804992676
Epoch 6 Loss: 0.8008065223693848
Epoch 7 Loss: 0.8183305263519287
Epoch 8 Loss: 0.8122203946113586
Epoch 9 Loss: 0.7897400856018066
Epoch 10 Loss: 0.7602871656417847
Epoch 11 Loss: 0.7324976325035095
Epoch 12 Loss: 0.7121672630310059
Epoch 13 Loss: 0.7013890743255615
Epoch 14 Loss: 0.6992337703704834
Epoch 15 Loss: 0.7030994296073914
Epoch 16 Loss: 0.7098110914230347
Epoch 17 Loss: 0.7163723707199097
Epoch 18 Loss: 0.7205312252044678
Epoch 19 Loss: 0.7211354374885559
```

## منابع و مراجع

<https://www.dailydoseofds.com/a-beginner-friendly-introduction-to-kolmogorov-/arnold-networks-kan>

<https://www.youtube.com/watch?v=qhQrRCJ-mVg>

[https://www.youtube.com/watch?v=7zpz\\_AlFW2w&t=12s](https://www.youtube.com/watch?v=7zpz_AlFW2w&t=12s)