پیاده‌سازی KAN

متن گزارش در محیط ژوپیتر محلی (local) به درستی نمایش داده می‌شود، ولی روی گوگل کولب مشکل دارد.

در ابتدا کتابخانه‌های مورد نیاز را ایمپورت می‌کنیم.

```
[ ]: import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import numpy as np
import pandas as pd
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import LabelEncoder
import torch.optim as optim
from sklearn.model_selection import train_test_split
```

اولین کلاسی که تعریف می‌کنیم نماینده‌ی مجموعه‌ای از توابع ϕ است که متناظر با یک نورون خروجی هستند. پس واضح است که هر یک از این مجموعه‌ها در هر لایه به تعداد ورودی‌های لایه تابع ϕ خواهد داشت و تعداد کل این مجموعه‌ها در هر لایه برابر تعداد نورون‌های خروجی لایه است.

هر یک از این مجموعه‌ها در ابتدا اندازه‌ی grid مربوط به اسپلاین‌ها را به همراه مرتبه‌ی اسپلاین‌ها و تابع پایه دریافت می‌کند. چون تعداد

هر یک از این مجموعه‌ها در ابتدا اندازه‌ی grid مربوط به اسپلاین‌ها را به همراه مرتبه‌ی اسپلاین‌ها و تابع پایه دریافت می‌کند. چون تعداد یال‌های ورودی از هر مجموعه برابر کل ورودی‌های لایه است و به ازای هر ویژگی دریافتی لایه مطابق صفحه‌ی ۱۰ مقاله، $G_1 + k$ تابع پایه‌ی بی‌اسپلاین داریم، ماتریس اولیه‌ی ضرایب `c` را با ابعاد $n \times (G_1 + k)$ ایجاد و مطابق پانویس صفحه‌ی ۶ مقاله، هرکدام را از توزیع نرمال با $\mu = 0$ و $\sigma = 0.1$ مقداردهی اولیه می‌کنیم. در ادامه n را تعداد ورودی‌های لایه و m را تعداد خروجی‌های آن در نظر می‌گیریم.

از طرفی مطابق آنچه که در صفحه‌ی ۶ آورده شده، به ازای هر تابع ϕ یک مقدار w_b و یک مقدار w_s داریم. پس باید بردار `self.w_s` و `self.w_b` را که به تعداد توابع ϕ مجموعه، یعنی تعداد ویژگی‌های ورودی آن عضو دارند ایجاد و مقداردهی اولیه کنیم. مطابق آنچه که در صفحه‌ی ۶ مقاله آمده $w_s = 1$ و لذا `w_s` برداری از 1ها خواهد بود و `w_b` نیز در پیاده‌سازی ما به صورت رندوم انتخاب می‌شود. همچنین دلیل استفاده از `nn.Parameter` روی متغیرهای `self.w_b`، `self.c` و `self.w_s` قابلیت آموزش‌پذیری آنها و در واقع بروزرسانی در فرآیند آموزش شبکه است.

متد `b_spline` با استفاده از رابطه‌ی بازگشتی زیر نوشته شده است:

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

که p درجه‌ی بی‌اسپلاین است.

تابع `b_splines` با دریافت بردار ورودی `x` که شامل ویژگی‌های یک نمونه است، ابتدا با استفاده از متد `b_spline` همه‌ی $G_1 + k$ بی‌اسپلاین ویژگی اول را محاسبه کرده و آنگاه با محاسبه‌ی بی‌اسپلاین‌ها برای ویژگی‌های دیگر، آنها را در پایین بی‌اسپلاین‌های ویژگی اول قرار داده و ماتریسی نهایی با ابعاد $n \times (G_1 + k)$ را خروجی می‌دهد.

طبق رابطه‌ی 2.12 مقاله که در صفحه‌ی ۶ آمده، متد `spline` را به گونه‌ای تعریف می‌کنیم که به ازای همه‌ی n ویژگی ورودی لایه، یعنی هر x_i، $spline(x_i)$ را محاسبه کند. برای این کار کافی است ماتریس‌های `self.c` و `self.b_splines(x)` (که در اینجا `x` نمونه‌ای دلخواه است) را به صورت درایه به درایه در هم ضرب کرده و آنگاه درایه‌های هر سطر را با هم جمع کنیم و بردار نهایی را با تغییر اندازه به ابعاد $n \times 1$ خروجی دهیم.

برای پیاده‌سازی متد `forward` از رابطه‌ی 2.10 استفاده می‌کنیم. در واقع بردار خروجی `spline(x)` را در بردار ضرایب `self.w_s` به صورت درایه به درایه ضرب کرده و همین کار را برای `self.b(x)` و `self.w_b` نیز انجام داده و مقدار اسکالر حاصل از جمع مقادیر بردار نهایی را خروجی می‌دهیم.

```
[ ]: class PostActivationSet(nn.Module):
    def __init__(self, input_size, grid_size, k, b, grid):
        super().__init__()
        self.grid_size = grid_size
        self.k = k
        self.b = b()
        self.c = nn.Parameter(torch.normal(0, 0.1, size=(input_size, grid_size + k)))
        self.w_b = nn.Parameter(torch.ones(input_size))
        self.w_s = nn.Parameter(torch.rand(input_size))
        self.grid = grid

    def b_spline(self, t, i, p=0):
        if p == 0:
            return ((self.grid[i] <= t) & (t < self.grid[i + 1])).int()
        return ((t - self.grid[i]) / (self.grid[i + p] - self.grid[i])) * self.b_spline(t, i, p - 1) + ((self.grid[i + p + 1] - t) / (self.grid[i + p + 1

    def b_splines(self, x):
        bases = torch.tensor([self.b_spline(x[0].item(), i) for i in range(self.grid_size + self.k)])
        for t in x[1:]:
            bases = torch.cat((bases, torch.tensor([self.b_spline(t.item(), i) for i in range(self.grid_size + self.k)])))
        return bases.view(x.shape[0], -1)

    def spline(self, x):
        return torch.sum(self.c * self.b_splines(x), dim=1).view(1, -1)

    def forward(self, x):
        return torch.sum(self.w_b * self.b(x) + self.w_s * self.spline(x)).item()
```

در ادامه کلاس `Layer` را که در واقع همان لایه‌های شبکه هستند پیاده‌سازی می‌کنیم. این کلاس ویژگی‌های ورودی و خروجی لایه، اندازه‌ی grid یا همان G_1، مرتبه‌ی اسپلاین، تابع پایه و بازه‌ی grid را دریافت می‌کند. مقدار پیش فرض k همان طور که در صفحه‌ی ۷ مقاله ذکر شده برابر 3 و بازه‌ی grid طبق بررسی پیاده‌سازی‌های دیگر KAN برابر $(-1, 1)$ قرار داده شده است.

در ادامه باید خود grid را ایجاد کنیم، چراکه یکی از ورودی‌های کلاس قبلی همین grid است. برای ایجاد grid با اندازه‌ی G_1 در بازه‌ای مثل $[t_0 = a, t_{G_1} = b]$ در ابتدا فاصله‌ی بین هر دو نقطه‌ی متوالی میان a و b را از رابطه‌ی زیر به دست می‌آوریم:

$$\delta = \frac{b - a}{G_1}$$

حال مطابق آنچه که در صفحه‌ی ۱۰ مقاله آورده شده، نقاط $t_{-1}, \dots, t_{-k}$ را از سمت چپ و نقاط $t_{G_1}, \dots, t_{G_1+k}$ را از سمت راست باید به grid مورد نظر بیفزاییم به طوری که فاصله‌ی بین هر نقطه‌ی متوالی جدید نیز همان δ باشد.

در نهایت باید مجموعه‌های توابع ϕ را به لایه اضافه کنیم. پس به تعداد خروجی‌های لایه مجموعه تشکیل می‌دهیم.

متد `forward` باید برداری را خروجی دهد که شامل خروجی‌های هر مجموعه از توابع ϕ است که به همین صورت هم پیاده‌سازی شده.

```
[ ]: class Layer(nn.Module):
    def __init__(self, num_input, num_output, grid_size, k=3, b=nn.SiLU, grid_range=(-1, 1)):
        super().__init__()
        self.num_input = num_input
        self.num_output = num_output
        grid_interval = (grid_range[1] - grid_range[0]) / grid_size + grid_range[0]
        grid = torch.arange(-k, grid_size + k + 1) * grid_interval + grid_range[0]
        self.post_activation_sets = nn.ModuleList()
        for i in range(num_output):
            self.post_activation_sets.append(PostActivationSet(num_input, grid_size, k, b, grid))

    def forward(self, x):
        out = torch.zeros(self.num_output)
        for i in range(self.num_output):
            out[i] += self.post_activation_sets[i](x)
        return out
```

کلاس `KAN` همان شبکه‌ی KAN و شامل لایه‌های آن است. این شبکه تعداد نورون‌های هر لایه را به همراه سایر پارامترهای مورد نیاز به شکل لیستی در ورودی دریافت می‌کند. متد `forward` هم ورودی شبکه را از لایه‌های متوالی عبور می‌دهد، اما چون پیاده‌سازی `forward` در کلاس‌های قبلی به صورت بچ بچ نبوده، این متد به ازای هرکدام از نمونه‌ها جداگانه نمونه‌های ورودی را از لایه‌ها عبور می‌دهد و در نهایت آنها را در کنار هم قرار می‌دهد. در انتها نیز چون دیتاستی که در ادامه با آن کار خواهیم کرد شامل برچسب‌های 0 و 1 است، خروجی نهایی شبکه را از تابع سیگموید عبور می‌دهیم.

```
[ ]: class KAN(nn.Module):
    def __init__(self, layer_sizes, grid_size, k=3, b=nn.SiLU, grid_range=(-1, 1)):
        super().__init__()
        self.layers = nn.ModuleList()
        for num_input, num_output in zip(layer_sizes, layer_sizes[1:]):
            self.layers.append(Layer(num_input, num_output, grid_size, k, b, grid_range))

    def forward(self, X):
        for layer in self.layers:
            unbinded_X = torch.unbind(X, dim=0)
            X = torch.stack([layer(x) for i, x in enumerate(X)], dim=0)
        return torch.sigmoid(X)
```

در ادامه می‌خواهیم شبکه را روی دیتاستی که از اینجا پیدا کرده‌ایم امتحان کنیم. لازم به ذکر است که کدهای قسمت آموزش و آزمایش از همین لینک آورده شده.

در ابتدا دیتاست را دانلود و از حالت فشرده خارج می‌کنیم:

```
[ ]: !wget https://archive.ics.uci.edu/static/public/151/connectionist+bench+sonar+mines+vs+rocks.zip
!unzip /content/connectionist+bench+sonar+mines+vs+rocks.zip
```

```
--2024-06-19 20:16:42--  https://archive.ics.uci.edu/static/public/151/connectionist+bench+sonar+mines+vs+rocks.zip
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified
Saving to: ‘connectionist+bench+sonar+mines+vs+rocks.zip.1’

connectionist+bench     [ <=>                ]  63.88K  --.-KB/s    in 0.02s

2024-06-19 20:16:42 (3.10 MB/s) - ‘connectionist+bench+sonar+mines+vs+rocks.zip.1’ saved [65413]

Archive:  /content/connectionist+bench+sonar+mines+vs+rocks.zip
replace sonar.all-data? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
replace sonar.mines? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
replace sonar.rocks? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
replace Index? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
replace sonar.names? [y]es, [n]o, [A]ll, [N]one, [r]ename: n
```

حال باید اطلاعات دیتاست را به صورت دیتافریم‌های مشخصی ذخیره کنیم. در اینجا عملیات انکود کردن و تبدیل دیتافریم‌های پانداس به پایتورچ نیز انجام شده است. همچنین دیتالودر را نیز با اندازه‌ی بچ ۱۶تایی ایجاد کرده و اطلاعات یک بچ را پرینت می‌کنیم.

```
[ ]: data = pd.read_csv("/content/sonar.all-data", header=None)
X = data.iloc[:, 0:60].values
y = data.iloc[:, 60].values

encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

loader = DataLoader(list(zip(X,y)), shuffle=True, batch_size=16)
for X_batch, y_batch in loader:
    print(X_batch, y_batch)
    break
```

```
tensor([[1.3500e-02, 4.5000e-03, 5.1000e-03, 2.8900e-02, 5.6100e-02, 9.2900e-02,
         1.0310e-01, 8.8300e-02, 1.5960e-01, 1.9080e-01, 1.5760e-01, 1.1120e-01,
         1.1970e-01, 1.1740e-01, 1.4150e-01, 2.2150e-01, 2.6580e-01, 2.7130e-01,
         3.8620e-01, 5.7170e-01, 6.7970e-01, 8.7470e-01, 1.0000e+00, 8.9480e-01,
         8.4200e-01, 9.1740e-01, 9.3070e-01, 9.0500e-01, 8.2280e-01, 6.9860e-01,
         5.8310e-01, 4.9240e-01, 4.5630e-01, 5.1590e-01, 5.6700e-01, 5.2840e-01,
         5.1440e-01, 3.7420e-01, 2.2820e-01, 1.1930e-01, 1.0880e-01, 4.3100e-02,
         1.0700e-01, 5.8300e-02, 4.6000e-03, 4.7300e-02, 4.0800e-02, 2.9000e-02,
         1.9200e-02, 9.4000e-03, 2.5000e-03, 3.7000e-03, 8.4000e-03, 1.0200e-02,
         9.6000e-03, 2.4000e-03, 3.7000e-03, 2.8000e-03, 3.0000e-03, 3.0000e-03],
        [2.6500e-02, 4.4000e-02, 1.3700e-02, 8.4000e-03, 3.0500e-02, 4.3800e-02,
         3.4100e-02, 7.8000e-02, 8.4400e-02, 7.7900e-02, 3.2700e-02, 2.0600e-01,
         1.9080e-01, 1.0650e-01, 1.4570e-01, 2.2320e-01, 2.0700e-01, 1.1050e-01,
         1.0780e-01, 1.1650e-01, 2.2240e-01, 6.8900e-02, 2.0600e-01, 2.3840e-01,
         9.0400e-02, 2.2780e-01, 5.8720e-01, 8.4570e-01, 8.4670e-01, 7.6790e-01,
         8.0550e-01, 6.2600e-01, 6.5450e-01, 8.7470e-01, 9.8850e-01, 9.3480e-01,
         6.9600e-01, 5.7330e-01, 5.8720e-01, 6.6630e-01, 5.6510e-01, 5.2470e-01,
         3.6840e-01, 1.9970e-01, 1.5120e-01, 5.0800e-02, 9.3100e-02, 9.8200e-02,
```

یک مدل با ورودی ۶۰تایی، دو لایه‌ی مخفی با ابعاد ۶۰تایی و ۳۰تایی و لایه‌ی خروجی تکی تعریف می‌کنیم:

```
[ ]: model = KAN([60, 60, 30, 1], 5)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
[9]: KAN(
       (layers): ModuleList(
         (0): Layer(
           (post_activation_sets): ModuleList(
             (0-59): 60 x PostActivationSet(
               (b): SiLU()
             )
           )
         )
         (1): Layer(
           (post_activation_sets): ModuleList(
             (0-29): 30 x PostActivationSet(
               (b): SiLU()
             )
           )
         )
         (2): Layer(
           (post_activation_sets): ModuleList(
             (0): PostActivationSet(
               (b): SiLU()
             )
           )
         )
       )
     )
```

تقسیم‌بندی داده‌های آموزشی و آزمایشی را با نسبت ۷۰ به ۳۰ انجام می‌دهیم. آنگاه دیتالودر را روی داده‌های آزمایشی ایجاد کرده و شروع به آموزش مدل می‌کنیم. تابع زیان را Binary Cross Entropy با توجه به ماهیت دودویی برچسب‌ها در نظر می‌گیریم. تعداد اپاک‌ها را در اینجا به تعداد خیلی کمی تعریف کرده‌ایم چراکه آموزش مدل زمان زیادی می‌برد و دقت کم مدل روی داده‌های آزمایشی هم احتمالاً به این موضوع مربوط است.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

loader = DataLoader(list(zip(X_train, y_train)), shuffle=True, batch_size=16)

n_epochs = 10
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.0001)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.requires_grad = True
        loss.backward()
        optimizer.step()

# evaluate accuracy after training
model.eval()
y_pred = model(X_test)
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```

```
Model accuracy: 49.21%
```

در ادامه می‌خواهیم مفهوم grid extension را هم که در مقاله آمده پیاده‌سازی کنیم. برای این کار باید طراحی خود را تغییر دهیم. در واقع ما در پیاده‌سازی قبلی داده‌ها را به صورت تکی به هر مجموعه از توابع ϕ می‌دادیم. اما در اینجا باید این کار را به صورت بچ بچ انجام دهیم. در ادامه به علت این موضوع می‌پردازیم.

مفهوم grid extension در اواخر صفحه‌ی ۹ و اوایل صفحه‌ی ۱۰ مقاله آمده است. در واقع ما تا اینجای کار سایز grid را برای بی‌اسپلاین‌ها برابر G_1 در نظر گرفتیم. اما این مقدار چندان زیادی نیست. به عنوان مثال در پیاده‌سازی‌هایی که مورد بررسی قرار دادیم، مقدار اولیه‌ی G_1 برابر 5 در نظر گرفته می‌شود. به همین دلیل نویسندگان برای افزایش دقت، افزودن نقاط کنترلی برای انعطاف‌پذیری بیشتر بی‌اسپلاین‌ها را پیشنهاد می‌کنند که در واقع در مراحلی از آموزش شبکه می‌توانیم سایز grid را از G_1 به G_2 افزایش دهیم. اما چنین کاری باعث می‌شود ضرایب جدیدی برای بی‌اسپلاین‌ها تعریف کنیم. نویسندگان تعریف این ضرایب را به گونه‌ای پیشنهاد می‌کنند که فاصله (نرم اختلاف) بین اسپلاین‌ها کمینه شود. به همین دلیل در صفحه‌ی ۱۰ مقاله استفاده از روش کمترین مربعات پیشنهاد شده است. اما کاهش فاصله باید نسبت به توزیع خاصی از داده‌ها صورت بگیرد، همان طور که در رابطه‌ی زیر نیز مشهود است:

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

پس باید داده‌ها را به صورت بچ بچ دریافت کرده تا ضرایب بی‌اسپلاین‌های متناظر با هر ویژگی نسبت به مقادیر نمونه‌های هر بچ گزینش شوند.

در ادامه تغییراتی را روی همان کلاس‌های قبلی انجام می‌دهیم. برای مثال ساخت grid را در متد جدید `create_grid` تعریف می‌کنیم تا هرگاه نیاز به بروزرسانی آن با اندازه‌ی جدید بود، همین متد را فراخوانی کنیم. از طرفی متد `b_splines` را از یکی از پیاده‌سازی‌های غیررسمی KAN آورده‌ایم، چراکه محاسبه‌ی بی‌اسپلاین‌ها برای هر ویژگی هر نمونه از یک بچ و قرار دادن آنها در یک تانسور سه بعدی پیچیده بوده و زمان زیادی می‌برد. این متد با محاسبات ماتریسی همین کار را ولی با کد کوتاه‌تری انجام می‌دهد. متد `spline` را نیز به گونه‌ای بازنویسی می‌کنیم که به ازای هر نمونه، ماتریس‌های `self.c` را همانند بشقاب‌هایی روی هم قرار دهد. در این صورت تانسوری با ابعاد $b \times n \times (G_i + k)$ ایجاد شده که همان ابعاد تانسور خروجی از متد `b_splines` را دارد و می‌توان با ضرب درایه به درایه‌ی این دو تانسور و جمع روی بعد دوم و تغییر ابعاد آن، به ماتریسی $b \times n$ رسید که همان $spline(x_{ij})$ها به ازای هر ویژگی iام نمونه‌ی jام بچ هستند. (در اینجا b همان تعداد نمونه‌های داخل هر بچ است.)

همچنین در پیاده‌سازی متدهای `forward` آرگومانی تحت عنوان `grid_extension` قرار می‌دهیم که در حالت پیش‌فرض نادرست است؛ اما در اپاک‌هایی که می‌خواهیم، مقدار درست را به آن می‌دهیم تا عملیات grid extension را انجام دهد. در صورت درست بودن این آرگومان، در همه‌ی مجموعه‌های شامل توابع ϕ، سایز grid دو برابر شده و هر سطر ماتریس جدید `self.c` به این صورت به دست می‌آید: روی بعد دوم تانسور حاصل از `b_splines(X)` حرکت کرده و هر ماتریس $b \times (G_2 + k)$ شامل بی‌اسپلاین‌های هر نمونه را استخراج می‌کنیم. آنگاه ستونی از ماتریس `spline(X)` که مربوط به همین مرحله است و متناظر با مقادیر اسپلاین‌های نمونه‌ها با سایز قبلی (در نتیجه با ابعاد $b \times 1$ است) را استخراج کرده و ضرایب بی‌اسپلاین‌های ویژگی متناظر با این مرحله را به دست می‌آوریم: به این صورت که جواب کمترین مربعات دستگاهی با ماتریس ضرایب به ابعاد $b \times (G_2 + k)$ و ماتریس ثوابت به ابعاد $b \times 1$ که برداری با ابعاد $(G_2 + k) \times 1$ خواهد بود را به دست آورده، تغییر شکل داده و در هر مرحله از n مرحله روی هم قرار می‌دهیم تا به ماتریس نهایی $n \times (G_2 + k)$ برسیم که همان `self.c` است.

```
[ ]: class PostActivationSet(nn.Module):
    def __init__(self, input_size, grid_size, k, b, grid_range):
        super().__init__()
        self.input_size = input_size
        self.grid_size = grid_size
        self.k = k
        self.b = b()
        self.c = nn.Parameter(torch.normal(0, 0.1, size=(input_size, grid_size + k)))
        self.w_b = nn.Parameter(torch.ones(input_size))
        self.w_s = nn.Parameter(torch.rand(input_size))
        self.grid_range = grid_range
        self.create_grid()

    def create_grid(self):
        grid_interval = (self.grid_range[1] - self.grid_range[0]) / self.grid_size + self.grid_range[0]
        self.grid = (torch.arange(-self.k, self.grid_size + self.k + 1) * grid_interval + self.grid_range[0]).expand(self.input_size, -1).contiguous()

    def b_splines(self, x: torch.Tensor):
        """
        Compute the B-spline bases for the given input tensor.

        Args:
            x (torch.Tensor): Input tensor of shape (batch_size, input_size).

        Returns:
            torch.Tensor: B-spline bases tensor of shape (batch_size, input_size, grid_size + spline_order).
        """
        assert x.dim() == 2 and x.size(1) == self.input_size

        grid: torch.Tensor = (
            self.grid
        )  # (in_features, grid_size + 2 * spline_order + 1)
        x = x.unsqueeze(-1)
        bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).to(x.dtype)
```

```
        for k in range(1, self.k + 1):
            bases = (
                (x - grid[:, : -(k + 1)])
                / (grid[:, k:-1] - grid[:, : -(k + 1)])
                * bases[:, :, :-1]
            ) + (
                (grid[:, k + 1 :] - x)
                / (grid[:, k + 1 :] - grid[:, 1:(-k)])
                * bases[:, :, 1:]
            )

        assert bases.size() == (
            x.size(0),
            self.input_size,
            self.grid_size + self.k,
        )
        return bases.contiguous()

    def spline(self, X):
        c = self.c.reshape(1, self.input_size, self.grid_size + self.k).expand(X.shape[0], self.input_size, -1)
        return torch.sum(c * self.b_splines(X), dim=2).reshape(X.shape[0], -1)

    def forward(self, X, grid_extension):
        if grid_extension:
            previous_spline = self.spline(X)
            self.grid_size *= 2
            self.create_grid()
            b_splines_ = self.b_splines(X)
            c = (torch.linalg.lstsq(b_splines_[:, 0, :], previous_spline[:, 0])).solution.reshape(1, -1)
            for i in range(1, self.input_size):
                c = torch.cat((c, (torch.linalg.lstsq(b_splines_[:, 0, :], previous_spline[:, 0])).solution.reshape(1, -1)))
            self.c = nn.Parameter(c)
        return torch.sum(self.w_b * self.b(X) + self.w_s * self.spline(X)).item()
```

[ ]:
```
class Layer(nn.Module):
    def __init__(self, num_input, num_output, grid_size, k=3, b=nn.SiLU, grid_range=(-1, 1)):
        super().__init__()
        self.num_input = num_input
        self.num_output = num_output
        grid_interval = (grid_range[1] - grid_range[0]) / grid_size + grid_range[0]
        grid = torch.arange(-k, grid_size + k + 1) * grid_interval + grid_range[0]
        self.post_activation_sets = nn.ModuleList()
        for i in range(num_output):
            self.post_activation_sets.append(PostActivationSet(num_input, grid_size, k, b, grid))

    def forward(self, X, grid_extension):
        out = torch.zeros(X.shape[0], self.num_output)
        for i in range(self.num_output):
            out[:, i] += self.post_activation_sets[i](X, grid_extension)
        return out
```

[ ]:
```
class KAN(nn.Module):
    def __init__(self, layer_sizes, grid_size, k=3, b=nn.SiLU, grid_range=(-1, 1)):
        super().__init__()
        self.layers = nn.ModuleList()
        for num_input, num_output in zip(layer_sizes, layer_sizes[1:]):
            self.layers.append(Layer(num_input, num_output, grid_size, k, b, grid_range))

    def forward(self, X, grid_extension=False):
        for layer in self.layers:
            X = layer(X, grid_extension)
        return torch.sigmoid(X)
```

ایجاد مدل و آموزش و ارزیابی را مشابه قبل انجام دادیم، ولی همان طور که انتظار می‌رفت به دلیل زمان زیادی که آموزش مدل صرف می‌کرد، در میانه‌ی کار آموزش را متوقف کردیم. در کد زیر در اپیاک‌های اول، پنجم و نهم grid extension رخ می‌دهد.

[ ]:
```
model = KAN([60, 60, 30, 1], 5)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

[13]:
```
KAN(
  (layers): ModuleList(
    (0): Layer(
      (post_activation_sets): ModuleList(
        (0-59): 60 x PostActivationSet(
          (b): SiLU()
        )
      )
    )
    (1): Layer(
      (post_activation_sets): ModuleList(
        (0-29): 30 x PostActivationSet(
          (b): SiLU()
        )
      )
    )
    (2): Layer(
      (post_activation_sets): ModuleList(
        (0): PostActivationSet(
          (b): SiLU()
        )
      )
    )
  )
)
```

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)

loader = DataLoader(list(zip(X_train, y_train)), shuffle=True, batch_size=16)

n_epochs = 10
loss_fn = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.0001)
model.train()
for epoch in range(n_epochs):
    for X_batch, y_batch in loader:
        if epoch % 4 == 0:
            y_pred = model(X_batch, True)
        else:
            y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.requires_grad = True
        loss.backward()
        optimizer.step()

model.eval()
y_pred = model(X_test)
acc = (y_pred.round() == y_test).float().mean()
acc = float(acc)
print("Model accuracy: %.2f%%" % (acc*100))
```