به نام خدا

تمرین سوم هوش محاسباتی

دستیاران آموزشی : مرتضی شهرابی _ غزل بخشنده

سارا سادات یونسی - ۹۸۵۳۳۰۵۳

فهرست

سوال ۱

چهار نورون با مختصات های [۰ ۰ ۰ ۱], [۰ ۱ ۱ ۰], [۰ ۰ ۰ ۱], [۱ ۱ ۰ ۰] در نظر بگیرید. تعداد نورون های خروجی را برابر با ۲ فرض کنید. می خواهیم این چهار نورون ورودی را در دو دسته، با استفاده از مدل Kohonenدسته بندی کنیم. این مدل را آموزش دهید. مقدار اولیه ی a را برابر با ۰,۵ در نظر بگیرید.)انجام تمام مراحل تا انتها لازم نیست. مراحل برای دوتا از نقاط انجام شود و روند کلی برای ادامه ی مراحل توضیح داده شود.

پاسخ

مدل Kohonen یک شبکه عصبی خودسازمانده است که برای خوشهبندی دادهها استفاده میشود. این شبکه دو لایه دارد: لایه ورودی و لایه خروجی. لایه ورودی شامل نورونهایی است که دادهها را نمایش میدهند و لایه خروجی شامل نورونهایی است که خوشهها را نمایش میدهند. هر نورون خروجی یک بردار وزن دارد که با دادههای ورودی مطابقت مییابد. هدف مدل Kohonen این است که بردارهای وزن را به گونهای تنظیم کند که هر خوشه شامل دادههایی باشد که به هم شبیهتر هستند.

برای آموزش مدل Kohonen، از الگوریتم یادگیری تطبیقی محلی (LCA) استفاده میشود. این الگوریتم به صورت زیر عمل میکند:

x_1	x_2	w_1	w_2
1	0	0.2	0.9
0	1	0.4	0.7
0	1	0.6	0.5
0	0	0.8	0.3

$d_j = \sum (w_{ij} - x_i)^2$

$\alpha = 0.5$

(1) دوره‌ی اول :

$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha (x_i - w_{ij}(\text{old}))$

$w_{12} = 0.9 + 0.5 \times (1 - 0.9) = 0.95$ | $w_{22} = 0.7 + 0.5 \times (0 - 0.7) = 0.35$

$w_{32} = 0.5 + 0.5 \times (0 - 0.5) = 0.25$ | $w_{42} = 0.3 + 0.5 \times (0 - 0.3) = 0.15$

$d_1 = (1 - 0.2)^2 + (0 - 0.4)^2 + (0 - 0.6)^2 + (0 - 0.8)^2 = 1.8$

$d_2 = (1 - 0.9)^2 + (0 - 0.7)^2 + (0 - 0.5)^2 + (0 - 0.3)^2 = 0.84$

(2) دوره‌ی دوم :

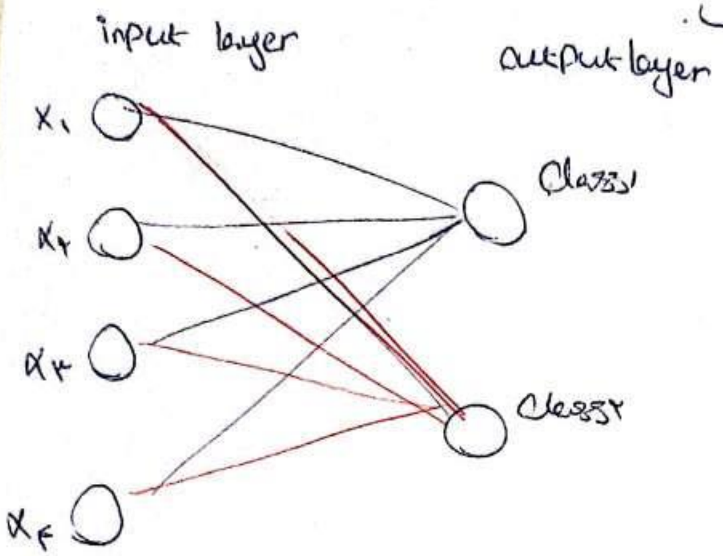
$w_{11} = 0.2 + 0.5 \times (0 - 0.2) = 0.1$ | $w_{21} = 0.4 + 0.5 \times (1 - 0.4) = 0.7$

$w_{31} = 0.6 + 0.5 (1 - 0.6) = 0.8$ | $w_{41} = 0.8 + 0.5 \times (0 - 0.8) = 0.4$

$d_1 = (0 - 0.2)^2 + (1 - 0.4)^2 + (1 - 0.6)^2 + (0 - 0.8)^2 = 1.2$

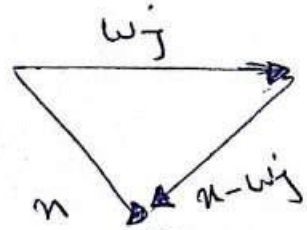
$d_2 = (0 - 0.95)^2 + (1 - 0.35)^2 + (1 - 0.25)^2 + (0 - 0.15)^2 = 1.91$

برای هر همسایه چک می‌شود و وزن‌ها با فاصله‌ی کمتر برنده می‌شود و این فرآیند تا زمانی ادامه پیدا می‌کند که وزن‌ها همگرا شوند زمانی که دو تکرار پشت سر هم وزن‌ها تغییر نکنند و دیگر تغییری نداشته باشیم یعنی همگرا شده است.



۴ ورودی داریم و
۲ کلاس خروجی (ابعاد ۲)
این مقادیر نورون‌ها را داریم.

۴ مرحله‌ی اصلی این الگوریتم



1) Initialization
2) رقابت
3) همکاری: یک نورون همسایه
4) Synaptic: دور کردن نورون‌های دوری.

$w_{ij}^{\text{جدید}} = w_{ij}^{\text{قدیم}} - \%5\,(x_i - w_{ij}^{\text{قدیم}})$

و همان کار را برای هر نمونه تا رسیدن به یک وزن همگرای مشخص و ثابت تکرار انجام می‌دهیم.

• یک داده ورودی را به صورت تصادفی انتخاب میکند.

• فاصله بین داده ورودی و بردارهای وزن هر نورون خروجی را محاسبه میکند. این فاصله میتواند با استفاده از معیارهای مختلفی مانند فاصله اقلیدسی یا فاصله مربعی محاسبه شود.

• نورون خروجی با کمترین فاصله را به عنوان برنده (winner) انتخاب میکند. این نورون بهترین مطابقت را با داده ورودی دارد.

• بردار وزن نورون برنده و نورونهای همسایهاش را به سمت داده ورودی حرکت میدهد. این کار با استفاده از یک نرخ یادگیری (a) و یک تابع همسایگی (h) انجام میشود. نرخ یادگیری مشخص میکند که چقدر بردار وزن تغییر کند و تابع همسایگی مشخص میکند که چقدر نورونهای همسایه تاثیر بپذیرند

مراحل الگوریتم SOM به شرح زیر است.

• ابتدا یک مجموعه داده ورودی و یک نقشه دوبعدی از نورونها را انتخاب میکنیم. میتوانیم اندازه، شکل و توپولوژی نقشه را بر اساس نیاز خود تعیین کنیم.

• سپس برای هر نورون، یک بردار وزن تصادفی با اندازهی مساوی با دادههای ورودی ایجاد میکنیم. این بردار وزن نشاندهندهی موقعیت نورون در فضای دادهها است.

• سپس یک دوره یا تکرار (iteration) را شروع میکنیم. در هر دوره، یک داده ورودی را به صورت تصادفی انتخاب میکنیم و به نقشه میفرستیم.

• سپس نورون برنده (winner neuron) یا نورون BMU (Best Matching Unit) را پیدا میکنیم. این نورون آن نورونی است که بردار وزن آن بیشترین شباهت را با داده ورودی دارد. میتوانیم از فاصله اقلیدسی یا هر معیار دیگری برای محاسبهی شباهت استفاده کنیم.

• سپس همسایگی نورون برنده را تعیین میکنیم. این همسایگی شامل نورونهایی است که در فاصلهی مشخصی از نورون برنده قرار دارند. میتوانیم از تابع همسایگی گوسی یا هر تابع دیگری برای محاسبهی فاصله و وزن همسایگی استفاده کنیم.

• سپس وزنهای نورون برنده و همسایگان آن را بهروز رسانی میکنیم. این بهروز رسانی بهگونهای انجام میشود که وزنها به سمت داده ورودی حرکت کنند و شباهت آنها با داده افزایش یابد. میتوانیم از فرمول زیر برای بهروز رسانی وزنها استفاده کنیم:

شرط توقف الگوریتم SOM میتواند بر اساس یکی از عوامل زیر باشد:

• تعداد تکرار (iteration) یا دوره (epoch) ثابت: در این حالت، الگوریتم پس از اجرای یک تعداد مشخص از تکرار یا دوره متوقف میشود. این روش ساده و راحت است، اما ممکن است منجر به جواب ناکامل یا غیربهینه شود

• تغییرات کم وزنها: در این حالت، الگوریتم پس از اینکه تغییرات وزنهای نورونها کمتر از یک حد آستانه شود، متوقف میشود. این روش نشاندهندهی رسیدن به یک حالت پایدار است، اما ممکن است منجر به جواب محلی یا گیر کردن در نقطهی ثابت شود

• تغییرات کم خطای بازسازی: در این حالت، الگوریتم پس از اینکه تغییرات خطای بازسازی دادهها کمتر از یک حد آستانه شود، متوقف میشود. خطای بازسازی معیاری است که نشاندهندهی اختلاف بین دادههای ورودی و دادههای بازسازی شده توسط نقشه است. این روش نشاندهندهی رسیدن به یک کیفیت خوب است، اما ممکن است منجر به جواب محلی یا گیر کردن در نقطهی ثابت شود

https://medium.com/machine-learning-researcher/self-organizing-map-som-c296561e2117

- Initialize the Network
- For i=1: **MaxIter**
  - **nr(t)** = Calculate neighbourhood radius (**Eq1**)
  - **σ(t)** = Calculate learning rate for epoch (**Eq2**)
  - For x =1: **All_inputs**
    - winner = Find **BMU** for x (**Eq3**)
    - for all w_k such that Distance(winner, w_k) < **nr(t)**
      - Adjust weights of w_k (**Eq4**)

    END

  END

END

Eq1 :-

$nr(t) = nr_0(t)\, e^{-(t/\lambda)}$ where λ = MaxIter / MaxNR

Eq2:-

$\sigma(t) = \sigma_0(t)\, e^{-(t/\lambda)}$ where λ = MaxIter / MaxNR

Eq3:-

$D(w\_i, w\_j) = \sum (w\_i_m , w\_j_m)^2$

Eq4:-

$w(t+1) = w(t) + \theta(t)\, \sigma(t)\, (x - w(t))$

where $\theta(t) = e^{-(\text{(distance from BMU^2) / (2 x nr(t)^2)})}$

$$\text{Euclidean distance} = \sqrt{(\text{observe value} - \text{centroid value})^2 + (\text{observe value} - \text{centroid value})^2}$$

$$\text{Euclidean distance} = \sqrt{((X_x - X_1))^2 + (X_y - Y_1)^2}$$

یک لایه ی ورودی و یک لایه ی خروجی داریم و نورون ورودی شامل ۴ نورون و نورون خروجی شامل دو کلاس و دو نورون است

سوال ۲

فرض کنید ورودی های xn , ...3,x 2,x 1,x قابل ذخیره کردن باشند. اگر مینیمم های محلی شبکه ی هاپفیلد دقیقا همین ورودی ها باشند، آیا لیست [۱،۱،-۱،-۱] ,[-۱،-۱،۱،۱] ,[۱،-۱،۱،-۱] ,[-۱،-۱،-۱،-۱] ,[۱،۱،۱،۱] قابل ذخیره سازی است؟ در صورتی که امکان پذیر نیست، دلیل خود را توضیح دهید و در غیر این صورت، عالوه بر توضیح علت امکان پذیر بودن، وزن های شبکه را محاسبه کنید.(۱۵ نمره)

پاسخ

بله قابل ذخیره سازی است چون اجزای بردارها در شبکه های هاپفیلد می توانند ۱+، -۱ یا ۱،۰ باشند

برای حل این سوال، ابتدا وزن متناسب با هر pattern را به دست آورده و سپس آن ها را جمع می کنیم .

$$w_{i,j}^{k} = x_i^k x_j^k$$

P1=(1,1,1,1)

I/J	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

P2=(-1,-1,-1,-1)

I/J	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

P3=(1, 1,-1,-1)

I/J	1	2	3	4
1	0	1	-1	-1

2	1	0	-1	-1
3	-1	-1	0	1
4	-1	-1	1	0

P4=(-1,- 1, 1, 1)

I/J	1	2	3	4
1	0	1	-1	-1
2	1	0	-1	-1
3	-1	-1	0	1
4	-1	-1	1	0

برای به دست آوردن وزن کل،[r] کافیست از رابطه زیر استفاده کنیم:

$$w_{i,j} = \sum_{k=1}^{k} w_{i,j}^{k}$$

I/J	1	2	3	4
1	0	4	0	0
2	4	0	0	0
3	0	0	0	4
4	0	0	4	0

حال، برای به دست آوردن انرژی هرpattern ، از رابطه زیر استفاده کنیم.

$$E = -\sum_{i,j} w_{i,j} o_i o_j$$

E(pattern1) = −(4 + 4 + 4 + 4) = −16

E(pattern2) = −(4 + 4 + 4 + 4) = −16

E(pattern3) = −(4 + 4 + 4 + 4) = −16

E(pattern4) = −(4 + 4 + 4 + 4) = −16

با توجه به مقادیر به دست آمده، نتیجه م۲گیریم م۲توانیم لیست داده شده را ذخیره کنیم؛ چرا که برای ذخیره سازی، کافیست پایین ترین سطح انرژی را داشته باشیم و برای ماتریس وزن بالا، پایین ترین سطح انرژی، ۱۶- می باشد.

نمایش به صورت جدولی :

چک می کنیم تا ببینیم پترن۱ ما پایدار هست و نقطه مینیموم انرژی است.

P3	1	2	3	4
T=0	1	1	1	1
T=1	1	1	1	1
T=2	1	1	1	1
…				
sigmaXiWij	4	4	4	4
	4	4	4	4

چک می کنیم تا ببینیم پترن2 ما پایدار هست و نقطه مینیموم انرژی است.

P2	1	2	3	4
T=0	1-	1-	1-	1-
T=1	1-	1-	1-	1-
T=2	1-	1-	1-	1-
…				
sigmaXiWij	4-	-4	4-	4-
	4-	-4	4-	4-

چک می کنیم تا ببینیم پترن3 ما پایدار هست و نقطه مینیموم انرژی است.

P3	1	2	3	4
T=0	1	1	-1	1-
T=1	1	1	-1	1-
T=2	1	1	-1	1-
…				
sigmaXiWij	4	4	4-	4-
	4	4	4-	4-

چک می کنیم تا ببینیم پترن4 ما پایدار هست و نقطه مینیموم انرژی است

P4	1	2	3	4
T=0	1-	-1	1	1
T=1	1-	-1	1	1
T=2	1-	-1	1	1
…				
sigmaXiWij	4-	4-	4	4
	4-	4-	4	4

حال ورودی جدید را به مدل وارد می کنیم. و پایدار بودن آن را از طریق روابط زیر بررسی میکنیم ✓ .در حل این سوال یه یک نکته توجه داریم: حاصل ضرب یک عدد مثبت در هر عددی عالمت آن را تغییر نمیدهد.

با توجه به این که ردیف با t = ۲ برابر شد و صورت سوال که ذکر کرده بود "مینیممهای محلی شبکهی هاپفیلد دقیقا همین ورودیها باشند." پس در t = ۰ شبکه پایدار بوده و به دنبال آن t=2نیز پایدار میشود. پس همه موارد لیست قابل ذخیره سازی میباشند .ماتریس وزن ها نیز در باالتر محاسبه شد

در سوالات تمرین مقدار حد آستانه را برابر با صفر قرار دادیم.

$$if\ a > 0\ then\ sign(ax) = sign(x)$$

$$x_1, x_2, x_3, x_4\ and\ x_i = \pm 1\ so\ sign(x_i) = x_i$$

$$u(i, t+1) = \sum_{j=1}^{N} w_{i,j}\, a(j,t)$$

$$a(i, t+1) = sign(u(i, t+1))$$

$$E(t) = -\sum_{i=1}^{N}\sum_{j=1}^{N} w_{i,j} a(i,t) a(j,t)$$

	1	2	3	4
$t = 0$	x_1	x_2	x_3	x_4
$t = 1$	$sign(4x_2) = x_2$	$sign(4x_1) = x_1$	$sign(4x_4) = x_4$	$sign(4x_3) = x_3$
$t = 2$	$sign(16x_1) = x_1$	$sign(16x_2) = x_2$	$sign(16x_3) = x_3$	$sign(16x_4) = x_4$

سوال ۳

یک شبکه عصبی MLP را پیاده سازی کرده و سپس آموزش دهید تا تابع y = x^۲را تخمین بزند. سپس خروجی شبکه عصبی را با خروجی تابع y = x^۲در محدوده [۳ -۳,] مقایسه کنید .

پاسخ

لایه ورودی :

شامل یک نورون ورودی

لایه پنهان :

یک لایه ی پنهان هر کدام شامل ۱۵ نورون

لایه خروجی :

شامل یک نورون که مقدار خروجی را نشان می دهد

تابع فعال سازی استفاده شده :

Relu

این تابع مناسب ترین تابع مورد استفاده می باشد زیرا دارای محدوده ی مشخصی نیست و همچنین می تواند تمام نمودار را پوشش دهد استفاده از توابع دیگر باعث تولید یک خط ثابت و صاف می شود.

از تابع relu برای تابع x^2 استفاده کردن ممکن است به چند دلیل مفید باشد:

- تابع relu محاسبه ی سریعی دارد و نیاز به تابع انفجاری یا تانژانت هذلولوی ندارد. این میتواند باعث کاهش زمان آموزش شبکه عصبی شود.

- تابع relu میتواند مشکل گرادیان محو شونده را حل کند که در توابع فعالسازی دیگر مانند سیگموئید وجود دارد. این مشکل زمانی رخ میدهد که مشتق تابع فعالسازی به سمت صفر میل کند و باعث کاهش سرعت یادگیری شبکه عصبی میشود. تابع relu این مشکل را با داشتن مشتق برابر با یک برای اعداد مثبت رفع میکند.

• تابع relu میتواند فعالبودن پراکندهای را در شبکه عصبی ایجاد کند که باعث افزایش تنوع و تمایزپذیری شبکه عصبی میشود. این به این معنی است که تنها بخشی از نورونها در یک زمان فعال هستند و بقیه صفر میشوند. این میتواند باعث کاهش اضافهبرازش شبکه عصبی شود.

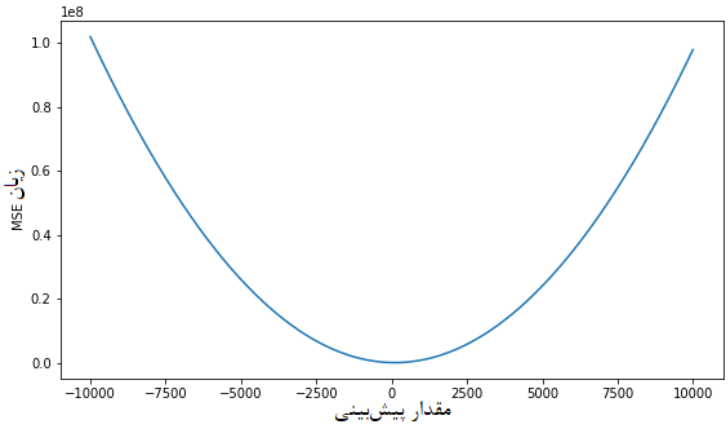
تابع خطا پیاده سازی شده :

توابعع خطای مختلف و کاربرد ان ها

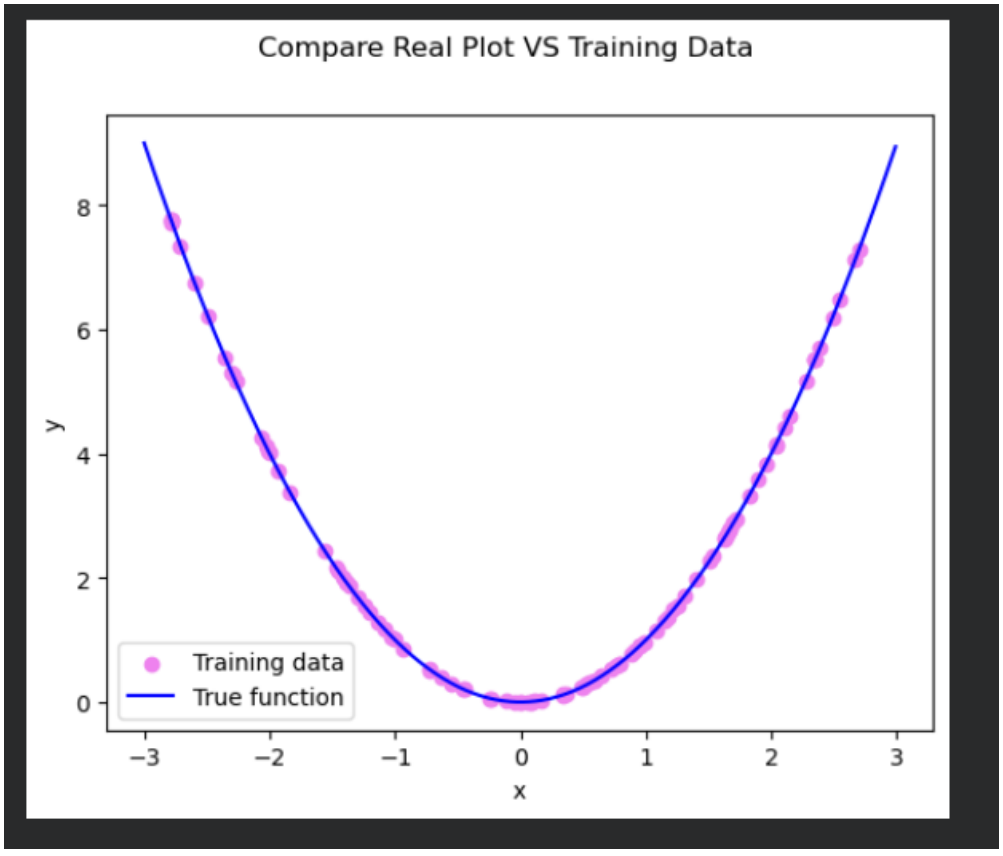
یکی از معروف‌ترین و معمول‌ترین توابع زیان در تحلیل رگرسیونی، میانگین مربعات خطا (Means Square Error) است که به اختصار MSE نامیده می‌شود. این تابع زیان، میانگین مربعات فاصله بین مقدار پیش‌بینی و واقعی را محاسبه می‌کند. شیوه و نحوه محاسبه آن در زیر دیده می‌شود:

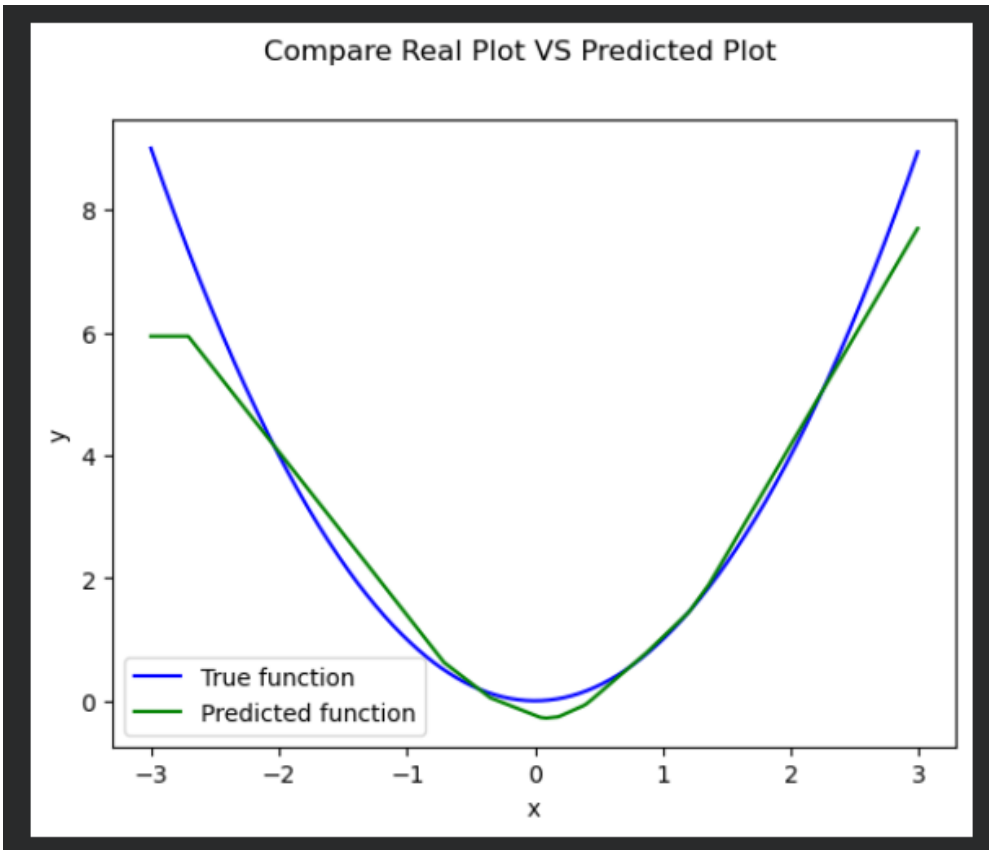
$$MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}$$

در مباحث آماری، معمولا به چنین تابعی، زیان L2گفته می‌شود. با توجه به استفاده از توان ۲ در محاسبهMSE ، شکل این تابع زیان برحسب مقدارهای پیش‌بینی (یا خطا) به صورت سهمی خواهد بود. فرض کنید که مقدار واقعی برای متغیر پاسخ (y) برابر است با صفر، در نتیجه نمودار تابع زیان MSE را براساس مقدارهای پیش‌بینی در محدوده ۱۰۰۰۰- تا ۱۰۰۰۰ می‌توان به صورت زیر رسم کرد.


1e8
1.0
0.8
0.6
0.4
0.2
0.0
MSE زیان
−10000
−7500
−5000
−2500
0
2500
5000
7500
10000
مقدار پیش‌بینی

نمودار ها :


Compare Real Plot VS Training Data
8
6
4
2
0
y
−3
−2
−1
0
1
2
3
x
Training data
True function



نتیجه گیری و مقایسه :

اگر توسط MLP یک تابع درجه دو را تخمین بزنیم، انگاه ممکن است تفاوتهایی با تابع اصلی داشته باشیم. این تفاوتها به عواملی مانند تعداد واحدهای پنهان، تابع فعالسازی، الگوریتم یادگیری، تعداد دادههای آموزشی و تنظیم پارامترهای فرعی (hyperparameters) بستگی دارند. به طور کلی، میتوان گفت که تخمین تابع درجه دو با MLP ممکن است دچار خطاهای زیر باشد :

• خطای تعمیمپذیری (generalization error): این خطا زمانی رخ میدهد که شبکه MLP بر روی دادههای آموزشی عملکرد خوبی داشته باشد، اما بر روی دادههای تست یا جدید عملکرد بدی داشته باشد. این خطا نشاندهندهی این است که شبکه MLP قادر به تعمیم دادن یادگیری خود به دادههای جدید نیست و فقط به دادههای آموزشی وابسته است. این خطا معمولاً ناشی از بیشبرازش (overfitting) است که زمانی رخ میدهد که شبکه MLP پیچیدگی بیش از حد داشته باشد و نویزها و جزئیات غیرمرتبط را هم یاد بگیرد. برای جلوگیری از این خطا، میتوان از روشهایی مانند اعتبارسنجی متقاطع (cross-validation)، جهتگیری متناهی (regularization)، تنظیم پارامترهای فرعی و افزایش تعداد دادههای آموزشی استفاده کرد

• خطای تقریب (approximation error): این خطا زمانی رخ میدهد که شبکه MLP قادر به یادگیری تابع درجه دو نباشد و از آن فاصله داشته باشد. این خطا نشاندهندهی این است که شبکه MLP پیچیدگی کافی برای تقریب تابع درجه دو را ندارد و نمیتواند الگوها و ویژگیهای مهم را شناسایی کند. این خطا معمولاً ناشی از کمبرازش (underfitting) است که زمانی رخ میدهد که شبکه MLP سادهتر از حد لازم باشد و نتواند پیچیدگی تابع درجه دو را درک کند. برای جلوگیری از این خطا، میتوان از روشهایی مانند افزایش تعداد واحدهای پنهان، تغییر تابع فعالسازی، تغییر الگوریتم یادگیری و کاهش جهتگیری متناهی استفاده کرد

برای کاهش خطای حاصل از پیشبینی ML، میتوان از روشهای مختلفی استفاده کرد. برخی از این روشها عبارتند از:

• انتخاب یک مدل مناسب برای مسئله: برای انتخاب مدل مناسب، باید ماهیت دادهها، توزیع دادهها، پیچیدگی مسئله، معیار ارزیابی و هدف پیشبینی را در نظر بگیریم. برای مثال، اگر مسئله ما رگرسیون خطی باشد، نمیتوانیم از مدلهای غیرخطی مانند شبکههای عصبی یا درخت تصمیم استفاده کنیم. همچنین، اگر دادههای ما دارای نویز یا انحراف زیاد باشند، باید از مدلهایی استفاده کنیم که دارای قابلیت تطبیق بالا و جهتگیری متناهی (regularization) باشند

• تقسیم دادهها به مجموعههای آموزش، توسعه و آزمون: برای ارزیابی عملکرد مدل و جلوگیری از بیشبرازش (overfitting) یا کمبرازش (underfitting)، باید دادهها را به سه مجموعه آموزش، توسعه و آزمون تقسیم کنیم. مجموعه آموزش برای یادگیری پارامترهای مدل، مجموعه توسعه برای تنظیم پارامترهای فرعی (hyperparameters) یا معماری مدل و مجموعه آزمون برای ارزیابی نهایی عملکرد مدل استفاده میشوند. معمولاً نسبت ۲۰/۲۰/۶۰ برای تقسیم دادهها به این سه مجموعه پیشنهاد میشود می شود

• افزایش تعداد و کیفیت دادهها: برای کاهش خطای پیشبینی ML، باید از دادههای کافی و معتبر برای آموزش مدل استفاده کنیم. اگر تعداد دادهها کم باشد، ممکن است مدل نتواند الگوها و ویژگیهای مهم را شناسایی کند و به دادههای آموزشی وابسته شود. اگر کیفیت دادهها پایین باشد، ممکن است مدل نتواند دادههای نویزی یا ناقص را تمیز کند و به دادههای غیرمرتبط یا اشتباه یاد بدهد. برای افزایش تعداد و کیفیت دادهها، میتوان از روشهایی مانند جمعآوری دادههای جدید، پیشپردازش دادهها، تکمیل دادههای گمشده، حذف دادههای پرت یا ناسازگار، تعادل برقرار کردن دادههای نامتوازن و افزایش دادهها (data augmentation) استفاده کنیم

• ارزیابی و مقایسه چندین مدل: برای کاهش خطای پیشبینی ML، باید چندین مدل را با هم مقایسه کنیم و بهترین مدل را بر اساس یک معیار ارزیابی انتخاب کنیم. معیار ارزیابی باید مناسب با نوع مسئله و هدف پیشبینی باشد. برای مثال، برای مسائل رگرسیون میتوان از معیارهایی مانند میانگین مربعات خطا (MSE)، ریشه میانگین مربعات خطا (RMSE)، میانگین مطلق خطا

(MAE) یا ضریب تعیین (R-squared) استفاده کرد. برای مسائل کلاسبندی میتوان از معیارهایی مانند دقت (accuracy)، صحت (precision)، بازخوانی (recall)، معیار F1 (F1-score) یا منحنی ROC (ROC curve) استفاده کرد

اگرچه mlp سریع است اما RBFبسیار دقیق تر است زیرا برای توابع غیر خطی و Classification های غیر خطی مدل های Radialعملکرد بهتری دارند به دلیل اینکه از متد های گاوسی استفاده میکنند.

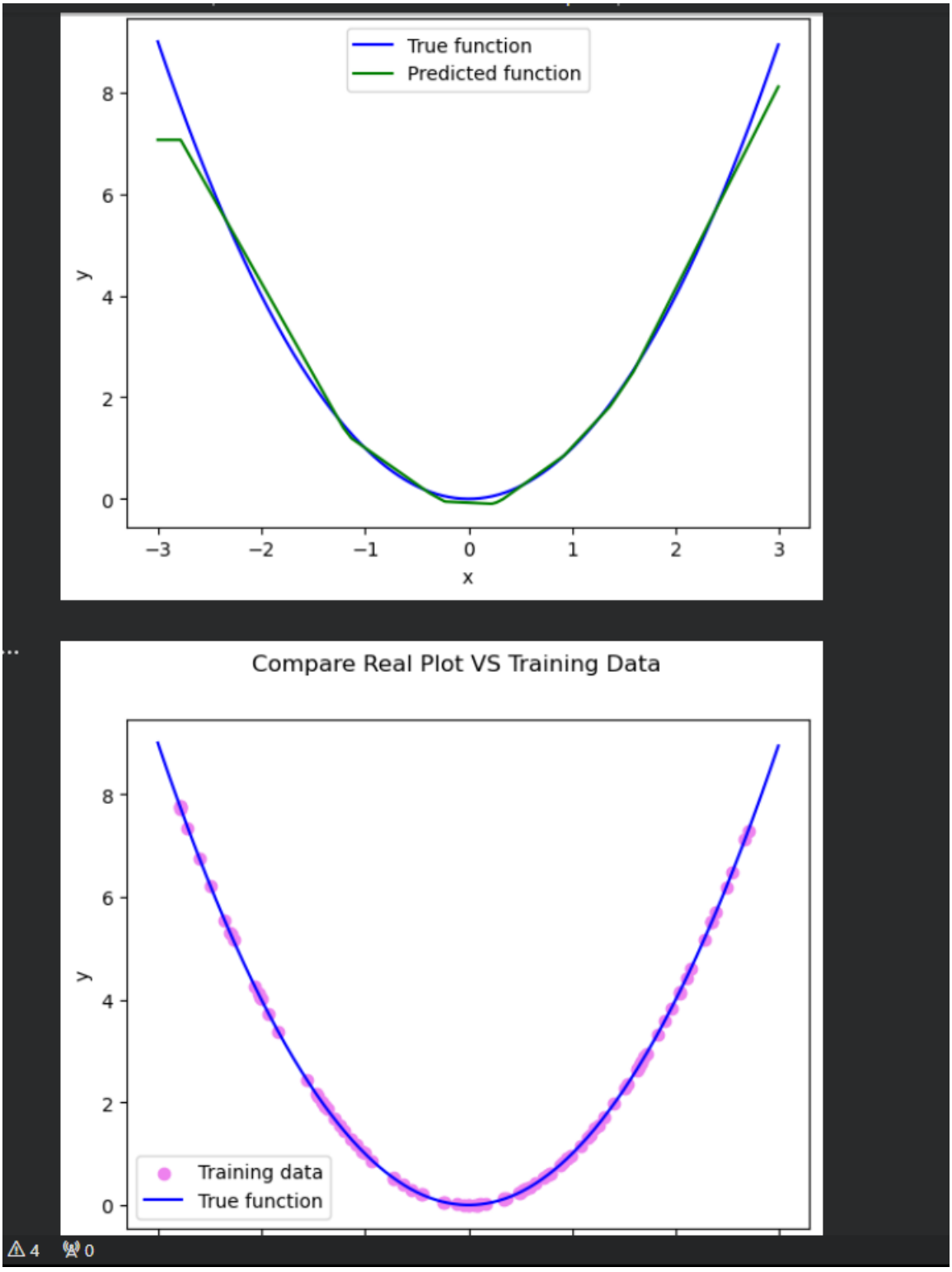
به طوری که اگر یکبار دیگر نورون های لایه ی پنهان ۲۰ و تعداد ایپاک ها ۳۰۰۰ شود تا حد بسیار خوبی روی نمودار فیت می شود چون :

طمئناً درک می شود که شبکه های عصبی اگر بخواهند مسائل را حتی به "ساده" x*x حل کنند، باید از پیچیدگی خاصی برخوردار باشند. و جایی که آنها واقعاً می درخشند زمانی است که با مجموعه داده های آموزشی بزرگ تغذیه می شوند.

روشی که هنگام تلاش برای حل چنین تقریب‌هایی از توابع انجام می‌شود، این نیست که فقط ورودی‌های (چند ممکن) را فهرست کرده و سپس به همراه خروجی‌های مورد نظر به مدل داده شود. به یاد داشته باشید، NN ها از طریق مثال ها یاد می گیرند و نه از طریق استدلال نمادین. و هر چه نمونه ها بیشتر باشد بهتر است. کاری که ما معمولاً در موارد مشابه انجام می دهیم، تولید تعداد زیادی نمونه است که متعاقباً به مدل برای آموزش می خوریم.

خوب، آنقدرها هم بد نیست! به یاد داشته باشید که NN ها تقریبگر تابع هستند: ما نباید انتظار داشته باشیم که آنها نه دقیقاً رابطه عملکردی را بازتولید کنند و نه "بدانند" که نتایج ۳ و -۳ باید یکسان باشند.

```
lr = 0.01
epochs =3000
input_data = 1
hidden_data =20
output_data = 1
```



در انتهای کد پیاده سازی خام توابع مورد نیاز با فانکشن های مختلف نیز انجام شده است.

توضیح کد :

```
import numpy as np
from matplotlib import pyplot as plt  # type: ignore
from matplotlib import cm  # type: ignore
from matplotlib import colors  # type: ignore
```
✓ 0.0s

```
# DATA TEST

test_data = np.arange(-3, 3, 0.01).reshape(-1, 1)
y = np.power(test_data,2)
```
✓ 0.0s

```
# PARAMETR AND HYPERPARAMETR

lr = 0.01
epochs =1500
input_data = 1
hidden_data =15
output_data = 1
```
✓ 0.0s

ین کد یک مجموعه داده آزمون از نوع numpy array ایجاد میکند که شامل ۶۰۰ نقطه داده است. این نقطهها از مقادیر -۳ تا ۳ با گام ۰,۰۱ تولید شدهاند و به صورت یک بردار ستونی با ابعاد ۶۰۰۱ قرار گرفتهاند. سپس یک بردار y از نوع numpy array ایجاد میکند که شامل مقادیر تابع درجه دو x^2 برای هر نقطه داده است. این بردار y نیز ابعاد ۶۰۰۱ دارد و میتواند به عنوان خروجی مورد نظر برای مجموعه داده آزمون در نظر گرفته شود. به عبارت دیگر، این کد یک مجموعه داده آزمون برای تخمین تابع درجه دو x^2 با ۶۰۰ نمونه از بازهی -۳ تا ۳ ایجاد میکند.

- lr یک هایپرپارامتر (hyperparameter) است که نشاندهندهی نرخ یادگیری (learning rate) برای آموزش یک شبکه عصبی است. این هایپرپارامتر تاثیر بسزایی بر روی سرعت و کیفیت یادگیری شبکه دارد و باید توسط کاربر تنظیم شود-https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/.

- epochs یک هایپرپارامتر است که نشاندهندهی تعداد دورهها (epochs) یا تکرارها (iterations) برای آموزش یک شبکه عصبی است. این هایپرپارامتر تاثیر بسزایی بر روی عملکرد و پایداری شبکه دارد و باید توسط کاربر تنظیم

- input_data یک پارامتر (parameter) است که نشاندهندهی تعداد واحدهای لایه ورودی (input layer) یک شبکه عصبی است. این پارامتر برابر با تعداد ویژگیهای دادههای ورودی است و باید بر اساس ماهیت دادهها مشخص

- hidden_data یک هایپرپارامتر است که نشاندهندهی تعداد واحدهای لایه پنهان (hidden layer) یک شبکه عصبی است. این هایپرپارامتر تاثیر بسزایی بر روی پیچیدگی و قدرت شبکه دارد و باید توسط کاربر تنظیم

- output_data یک پارامتر است که نشاندهندهی تعداد واحدهای لایه خروجی (output layer) یک شبکه عصبی است. این پارامتر برابر با تعداد خروجیهای مورد نظر برای پیشبینی است و باید بر اساس نوع مسئله مشخص

```
# REAL FUNCTION

def real(x):
    return np.power(x,2)

np.random.seed(20)
train_data = np.random.uniform(low=-3,high= 3 , size=(100,1))
y_train= real(train_data)
```
✓ 0.0s

```
# INITIAL W & B
W_1 = np.random.rand(input_data, hidden_data)
B_1n = np.zeros((1, hidden_data))

W_2 = np.random.rand(hidden_data, output_data)
B_2n = np.zeros((1, output_data))
```
✓ 0.0s

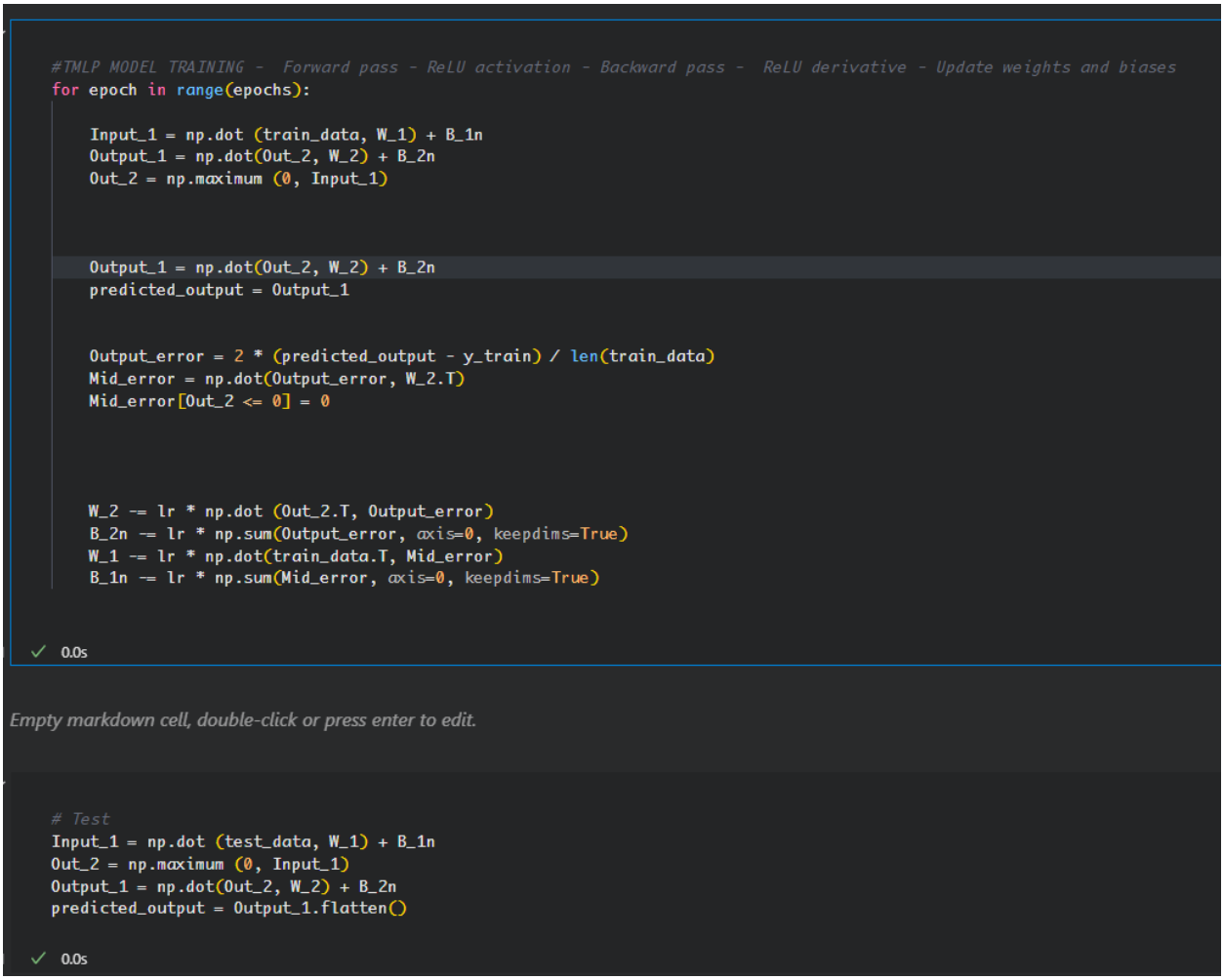
ین کد چند پارامتر و هایپرپارامتر را برای یک شبکه عصبی چندلایه (MLP) تعریف میکند. این شبکه عصبی دارای یک لایه ورودی، یک لایه پنهان و یک لایه خروجی است و قصد دارد یک تابع درجه دو را تخمین بزند. توضیح این پارامترها و هایپرپارامترها به شرح زیر است:

- W_1 و B_1n: اینها پارامترهای شبکه عصبی هستند که نشاندهندهی وزنها و بایاسهای لایه پنهان هستند. این پارامترها به صورت تصادفی با توزیع یکنواخت از مقادیر بین صفر و یک ایجاد شدهاند. اندازهی W_1 برابر با input_data * hidden_data و اندازهی B_1n برابر با ۱ * hidden_data است.

- W_2 و B_2n: اینها پارامترهای شبکه عصبی هستند که نشاندهندهی وزنها و بایاسهای لایه خروجی هستند. این پارامترها به صورت تصادفی با توزیع یکنواخت از مقادیر بین صفر و یک ایجاد شدهاند. اندازهی W_2 برابر با hidden_data * output_data و اندازهی B_2n برابر با ۱ * output_data است.

- real: این یک تابع است که تابع درجه دو x^2 را محاسبه میکند. این تابع برای مقایسهی خروجی شبکه عصبی با خروجی مورد نظر استفاده میشود.

- train_data و y_train: اینها مجموعه داده آموزشی هستند که برای یادگیری پارامترهای شبکه عصبی استفاده میشوند. این مجموعه داده شامل ۱۰۰ نمونه از بازهی -۳ تا ۳ است که به صورت تصادفی با توزیع یکنواخت ایجاد شدهاند. y_train برابر با مقادیر تابع درجه دو برای هر نمونه است.



```
#TMLP MODEL TRAINING -  Forward pass - ReLU activation - Backward pass -  ReLU derivative - Update weights and biases
for epoch in range(epochs):

    Input_1 = np.dot (train_data, W_1) + B_1n
    Output_1 = np.dot(Out_2, W_2) + B_2n
    Out_2 = np.maximum (0, Input_1)



    Output_1 = np.dot(Out_2, W_2) + B_2n
    predicted_output = Output_1


    Output_error = 2 * (predicted_output - y_train) / len(train_data)
    Mid_error = np.dot(Output_error, W_2.T)
    Mid_error[Out_2 <= 0] = 0




    W_2 -= lr * np.dot (Out_2.T, Output_error)
    B_2n -= lr * np.sum(Output_error, axis=0, keepdims=True)
    W_1 -= lr * np.dot(train_data.T, Mid_error)
    B_1n -= lr * np.sum(Mid_error, axis=0, keepdims=True)


✓ 0.0s
```

Empty markdown cell, double-click or press enter to edit.

```
# Test
Input_1 = np.dot (test_data, W_1) + B_1n
Out_2 = np.maximum (0, Input_1)
Output_1 = np.dot(Out_2, W_2) + B_2n
predicted_output = Output_1.flatten()

✓ 0.0s
```

این کد یک الگوریتم یادگیری برای یک شبکه عصبی چندلایه (MLP) است که قصد دارد یک تابع درجه دو را تخمین بزند. این الگوریتم از روش گرادیان نزولی (gradient descent) برای بهینهسازی پارامترهای شبکه استفاده میکند. این کد مراحل زیر را انجام میدهد:

- برای هر دوره (epoch) یا تکرار (iteration)، دادههای آموزشی را به شبکه عصبی میفرستد و خروجی شبکه را محاسبه میکند. این خروجی برابر با predicted_output است.

- سپس خطای پیشبینی شبکه را با مقایسهی خروجی شبکه با خروجی مورد نظر (y_train) محاسبه میکند. این خطا برابر با Output_error است.

- سپس گرادیان خطا را نسبت به پارامترهای شبکه (W_1, B_1n, W_2, B_2n) محاسبه میکند. این گرادیان نشاندهندهی جهتی است که باید پارامترها را در آن تغییر دهیم تا خطا کاهش یابد. این گرادیان برابر با مقادیری است که از پارامترها کم میشوند.

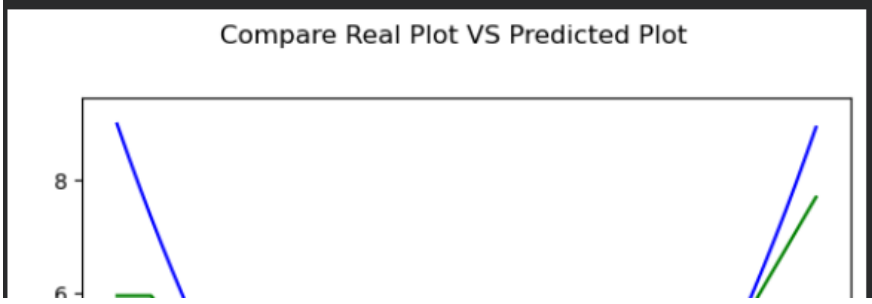
- سپس پارامترهای شبکه را با استفاده از نرخ یادگیری (lr) و گرادیان خطا بهروز رسانی میکند. این بهروز رسانی بهگونهای انجام میشود که پارامترها به سمت مینیمم محلی خطا حرکت کنند.

- این فرآیند را تا زمانی که شرط توقف برقرار شود، ادامه میدهد. شرط توقف میتواند بر اساس تعداد دورهها، تغییرات خطا یا هر معیار دیگری باشد.

- در نهایت، برای ارزیابی عملکرد شبکه، دادههای آزمون (test_data) را به شبکه میفرستد و خروجی شبکه را محاسبه میکند. این خروجی برابر با predicted_output است که میتواند با خروجی مورد نظر (y_test) مقایسه شود.

```
plt.plot(test_data, real (test_data), label='True function', linestyle='solid', color='blue')
plt.plot(test_data, predicted_output, label='Predicted function', linestyle='solid', color='green')
plt.xlabel('x')
plt.ylabel('y')
plt.suptitle('Compare Real Plot VS Predicted Plot')
plt.legend()
plt.show()

plt.scatter(train_data, y_train, label='Training data' , color='violet')
plt.plot(test_data, real (test_data), label='True function', linestyle='solid', color='blue')
plt.xlabel('x')
plt.ylabel('y')
plt.suptitle('Compare Real Plot VS Training Data')
plt.legend()
plt.show()
```

✓ 0.2s



و بعد رسم نمودار و محاسبه خطا و مقدار لاس با mSE

```
loss = np.mean((Out_2-y_train)**2)
```

https://stackoverflow.com/questions/55170460/neural-network-for-square-x2-approximation

سوال ۴

یک شبکه هاپفیلد طراحی کنید تا با شروع از رشته بیتی ۰۱۰۰۰۰، رشته بیتی ۱۱۱۱۰۰ را شناسایی کند، یا به عبارت دیگر به آن همگرا باشد. ماتریس وزن ها و جدول محاسبات را ارائه کنید.

پاسخ

1	0	1	1	-1	-1
0	1	0	0	0	-1
1	0	1	1	-1	0
0	1	0	1	-1	-1
0	1	-1	-1	0	-1
-1	-1	1	0	-1	0

	X1	X2	X3	X4	X5	X6
T=0	0	1	0	0	0	0
T=1	1	1	1	1	1	0
T=2	**1**	**1**	**1**	**1**	**0**	**0**
T=3	**1**	**1**	**1**	**1**	**0**	**0**
$\sum_{i}^{n} xiwij$						
	0	1	0	1	1	-1
	2	1	2	1	-1	-2
	3	1	3	2	-1	-1

آستانه را با ۰ قرار می دهیم مانند اسلاید ها

می خواهیم به مقدار ۱۱۱۱۰۰ همگرا شویم در نتیجه جدول اول را وزن دهی را طوری تنظیم کردیم که دو ستون و ردیف را شامل اعداد – و بقیه این ها مثبت قرار می دهیم.

و بعد از محاسبه می بینیم در t=2 به جواب مورد نظر همگرا شدیم.

شبکه ما در شبکه های دیگر یکسری ورودی و خروجی داشتیم که می دادیم به شبکه عصبی و با آن آموزش می دیدیم در هاپفیلد یکسری الگو الگو باید بدهیم تا آن بتواند با استفاده از آن الگوها سیستم را به آن الگو برساند. توجه شود که در هاپفیلد تمام ارتباطات در شبکه هاپفیلد دوطرفه و متقارن هستند برای شروع نیاز به ۶ نورون داریم.

برای طراحی یک شبکه هاپفیلد که با شروع از رشته بیتی ۰۱۰۰۰۰ رشته بیتی ۱۱۱۱۰۰ را شناسایی کند، میتوانیم از مراحل زیر اقدام کنیم:

- ابتدا باید ماتریس وزن شبکه را با استفاده از قانون یادگیری هب تعیین کنیم. این قانون بیان میکند که وزن اتصال بین دو نورون i و j برابر با حاصلضرب مقادیر نورونها در حالت ذخیره شده است. یعنی:

$$w_{ij} = v_i v_j$$

میتوانیم مقادیر سطر اول را با رشته بیتی اولیه ۰۱۰۰۰۰ مقداردهی کنیم. سپس میتوانیم مقادیر سطر دوم را با استفاده از قانون فعالسازی محاسبه کنیم. برای مثال، برای محاسبه داریم:

$$v_1(1) = sgn(\sum_j w_{1j} v_j(0)) =$$

$$sgn(w_{11} v_1(0) + w_{12} v_2(0) + w_{13} v_3(0) + w_{14} v_4(0) + w_{15} v_5(0) + w_{16} v_6(0)) =$$

$$sgn(1 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 0 + 0 \times 0 + 0 \times 0) = sgn(1) = 1$$

- سپس باید جدول محاسبات شبکه را با استفاده از قانون فعالسازی تعیین کنیم. این قانون بیان میکند که مقدار نورون i در هر مرحله برابر با علامت مجموع وزنشده مقادیر نورونهای دیگر است. یعنی:

$$v_i(t+1) = sgn(\sum_j w_{ij} v_j(t))$$

به همین ترتیب میتوانیم مقادیر بقیه نورونها را در سطر دوم محاسبه کنیم. میبینیم که در سطر دوم، رشته بیتی ۱۱۱۱۰۰ به دست میآید که همان حالت ذخیره شده است. این بدان معنی است که شبکه هاپفیلد به این حالت همگرا شده است و دیگر تغییری نمیکند. بنابراین، نیازی به محاسبه سطرهای بعدی نیست و جدول محاسبات تمام میشود.

سوال ۵

مسئله فروشنده دوره گرد یا Problem Salesman Traveling را در نظر بگیرید . در این مسئله تعداد ی شهر داریم و هزینه ی رفتن مستقیم از یکی به دیگری را می دانیم. مطلوب است کم هز ینه ترین مسیر ی که از یک شهر شروع م ی شود و از تمام ی شهر ها دقیقا یکبار عبور کند و به شهر شروع بازگردد. توضیح دهید این مسئله با کدام یک از شبکه های عصب ی که تاکنون خواندهاید قابل حل است. در صورتی که این مسئله با شبکه ای قابل حل بود، الگوریتم، سا\pختار شبکه و سایر توضیحات را ارائه دهید . در صورت غیر قابل حل بودن نیز دلیل خود را توضیح دهید .

Hopfield , SOM, MLP

پاسخ

مسئله فروشنده دوره گرد یک چالش شناخته شده در علوم کامپیوتر است. این مسئله شامل یافتن کوتاه ترین مسیر ممکن است که همه شهرها را در یک نقشه معین فقط یک بار طی میکند و N- complete است : .

MLP : حل مسئله با روش MLP امکانپذیر نیست، زیرا این یک مسئله ی unsupervised است و برای حل کردن آن باید یک لیبل برای هر شهر از قبل بدانیم اما اطلاعی از جایگاه شهرها در مسیر خود نداریم .

Hopfield: با توجه به لینک این مقاله این مسئله با شبکه ی هاپفیلد قابل حل شدن است. یک شبکه با تعداد nنورون میسازیم که n تعداد شهرهاست. سپس وزنهای بین نورونها را طبق فواصل بین آن ها مقداردهی اولیه میدهیم. در انتهای آموزش اگر وزن هر کدام از نورونهای شبکه در جایگاه مختلف یک شود بهترین مسیر است.

https://arxiv.org/ftp/arxiv/papers/2202/2202.13746.pdf#:~:text=The%20Travelling%20Salesmen%20problem%20has,this%20problem%20is%20Djikstra's%20Algorithm.

تمام توضیحات نیز در این مقاله آمده است.

همچنین یک مقاله دیگر در همین رابطه در فایل پیوست شده.

شبکه عصبی هاپفیلد در سال ۱۹۸۵، هاپفیلد شبکه کاملاً متصل را طراحی کرد که بعدها به عنوان شبکه عصبی هاپفیلد شناخته شد [۱۴]. او 10 TSP شهر و ۳۰ شهر را شبیه سازی کرد. علاوه بر این، این اولین الگوریتمی بود که از عصبی استفاده کرد شبکه برای حل مشکل بهینه سازی TSP. همانطور که در شکل ۱ نشان داده شده است، ایده الگوریتم تبدیل آن است تابع هدف وارد تابع انرژی شبکه عصبی می شود و تابع انرژی را در فرآیند به حداقل می رساند اجرای شبکه عصبی به منظور دستیابی به راه حل بهینه محلی. هاپفیلد تابع انرژی را تعریف کرد و معادلات حرکت مسئله TSP در معادلات. (۵) و (۶)

در اینجاA ، Bو C همگی مثبت هستند، که معادل سه عبارت جریمه به اضافه تابع هدف است .در اینجا یک شبکه عصبی هاپفیلد دو لایه در شکل ۱ آمده است. لایه صفر ورودی شبکه است و اولین لایه لایه گره نورون است .ورودی عصبی شامل ورودی خارجی و بازخورد عصبی است خروجی در معادله (7)اینجا $wwiiii$ وزن است، $yyii = [\ xx1$، $xx2$، ⋯، $xxNN]$ورودی خارجی است .تفاوت شبکه هاپفیلد با سایر شبکه های عصبی در این است که وزن شبکه آن نیست از طریق یادگیری مکرر تعیین می شود، اما به طور مستقیم داده می شود، و سپس وضعیت شبکه به روز می شود با توجه به معادله عملیاتی شبکه و در نهایت به جواب بهینه محلی می رسد .هاپفیلد دریافت که این شبکه عصبی برای مشکلات کوچکتر از ۳۰ شهر بسیار خوب عمل می کند .با این حال، اعمال نشد زمانی که TSP بزرگتر از ۳۰ شهر باشد .اگرچه شبکه هاپفیلد تقریباً به طور کامل TSPرا حل می کند، اما کاستی هایی نیز دارد که وجود داشته است توسط بسیاری از محققان بهینه شده است .لو [۱۵] ترم سوم تابع انرژی را با توجه به ضعیف بودن آن بهبود بخشید همگرایی؛ علاوه بر این، Qiaoو همکاران [۱۶] نیز تابع انرژی را با افزودن یک عبارت تصحیح به عملکرد انرژی و ایجاد یک شبکه عصبی بی نظم سینوسی تبدیل فرکانس نویز هیسترتیک جدید (HNFCSCNN)برای حل TSP که نتیجه رضایت بخشی دریافت کرد .لی و همکاران [۱۷] وزن اتصال را با توجه به کمبود آن بهبود بخشیدند که به راحتی در محلی قرار گرفت .کمترین .آنها وزن اتصال را بر اساس عملکرد تابع هدف تغییر دادند .گارسیا [18] روش تقسیم و غلبه را برای بهبود عملکرد شبکه هاپفیلد در TSP اتخاذ کرد

$$E = \frac{A}{2}\sum_{x=1}^{N}\sum_{i=1}^{N}\sum_{j=1}^{N} X_{xj}X_{xj} + \frac{B}{2}\sum_{i=1}^{N}\sum_{x=1}^{N}\sum_{y=x}^{N} X_{xi}X_{yi} + \frac{C}{2}\left(\sum_{x=1}^{N}\sum_{i=1}^{N} X_{xi} - N\right)^2$$
$$+\frac{D}{2}\sum_{x=1}^{N}\sum_{y=1}^{N}\sum_{i=1}^{N} d_{xy}X_{xi}\left(X_{y,i+1} + X_{y,i-1}\right) \quad (5)$$

$$\frac{dU_{xi}}{dt} = -\frac{\partial E}{\partial X_{xi}} = -A\left(\sum_{i=1}^{N} X_{xi} - 1\right) - B\left(\sum_{y=1}^{N} X_{yi} - 1\right) - D\sum_{y=1}^{N} d_{xy}X_{y,i+1} \quad (6)$$

$$z_j = \sum_{i}^{N} w_{ij}y_i + x_j \quad (7)$$

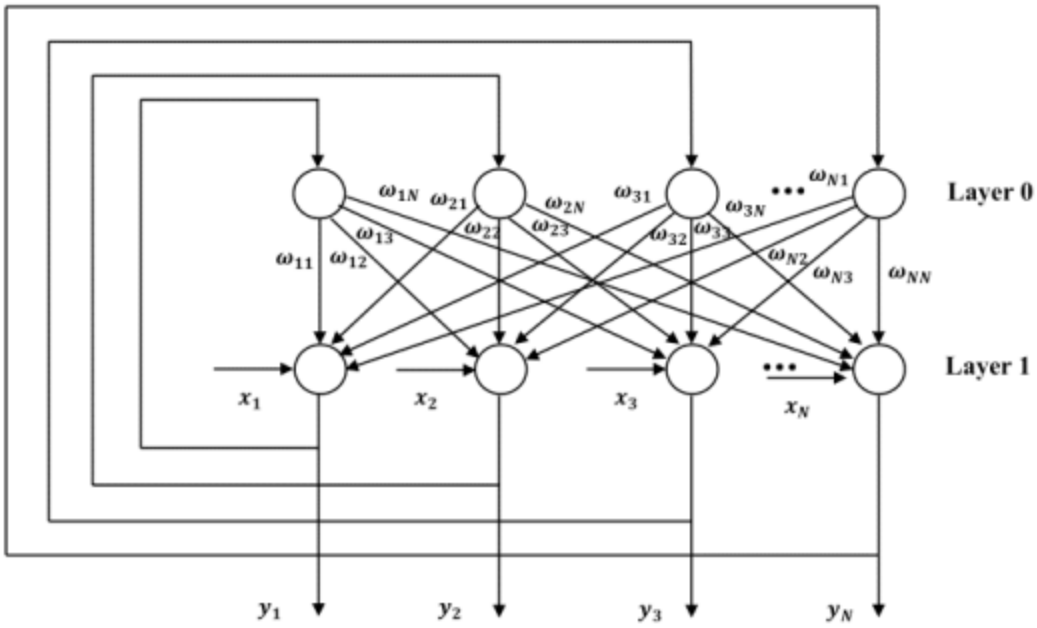


Fig. 1. Two layer of Hopfield network

• شبکه هاپفیلد یک شبکه عصبی بازگشتی است که از واحدهای دو حالته تشکیل شده است. این واحدها میتوانند مقادیر ۱ یا -۱ را به خود بگیرند و بر اساس مجموع ورودیهایی که از سایر واحدها دریافت میکنند،

حالت خود را تغییر میدهند. این واحدها به صورت متقارن به هم متصل هستند و وزن اتصالات از طریق قانون یادگیری هب (Hebb) تنظیم می شوند

• برای حل مسئله فروشنده دوره گرد با شبکه هاپفیلد، ابتدا باید یک ماتریس مجاورت از هزینههای مسیر بین شهرها را تهیه کنیم. سپس باید یک ماتریس وزن از اندازه n*n را ایجاد کنیم که n تعداد شهرها است. این ماتریس وزن را میتوان به صورت زیر محاسبه

• سپس باید یک بردار حالت از اندازه n را ایجاد کنیم که نشاندهنده حالت فعلی شبکه است. این بردار حالت را میتوان به صورت تصادفی یا با توجه به یک مسیر اولیه مقداردهی کرد.

که به معنی این است که شهر اول انتخاب شده است و سایر شهرها انتخاب نشدهاند.

• سپس باید بردار حالت را به روز رسانی کنیم تا به یک حالت پایدار برسیم. برای این کار، میتوان از روشهای مختلفی مانند به روز رسانی همزمان، به روز رسانی ترتیبی یا به روز رسانی تصادفی استفاده کرد. در هر روش، باید مجموع ورودیهای هر واحد را با آستانهای مقایسه کنیم و بر اساس نتیجه، حالت واحد را تغییر دهیم. برای مثال، اگر از روش به روز رسانی همزمان استفاده کنیم، باید بردار حالت را به صورت زیر به روز رسانی کنیم.

پیاده سازی با هاپفیلد در فایل موجود است .

SOM : ازآنجا که SOM داده ها با ویژگی ها شبیه هم را در نزدیکی هم قرار میدهد میتوان از این روش برای مینیمم کردن فاصله ها استفاده کرد .برای استفاده از این شبکه برای حل TSP ،مفهوم اصلی درک نحوه تغییر تابع همسایگی است. اگر به جای یک شبکه، یک آرایه دایرهای از نورون ها را اعلام کنیم، هر گره فقط از نورون های جلو و پشت خود آگاه خواهد بود. یعنی شباهت درونی فقط در یک بعد کار خواهد کرد. با انجام این اصلاح جزئی، نقشه ی خودسازماندهی مانند یک حلقه الاستیک رفتار میکند و به شهرها نزدیکتر میشود اما به لطف عملکرد همسایگی تلاش میکند تا محیط آن را به حداقل برساند. اگرچه این اصلاح ایده اصلی پشت این تکنیک است، اما آنطور که هست کار نخواهد کرد. الگوریتم به سختی همگرا می شود. برای اطمینان ن از همگرایی آن، میتوانیم نرخ یادگیری α را برای کنترل کاوش و بهرهبرداری از الگوریتم در نظر بگیریم. برای به دست آوردن اکتشاف بالا در ابتدا، و پس از آن بهره برداری بالا در اجرا، ما باید هم در تابع همسایگی و هم در نرخ یادگیری یک کاهش را لحاظ کنیم. کاهش سرعت یادگیری، جابجایی تهاجمی کمتری از نورونهای اطراف مدل را تضمین میکند و تضعیف همسایگی منجر به بهره برداری متوسطتر از مینیمم محلی هر بخش از مدل میشود .نقشه ی خود سازمانده ی)یا نقشه kohonen یا(SOM نوعی شبکه عصبی مصنوعی است که همچنین از مدلها ی

بیولوژیکی سیستمهای عصبی دهه ۱۹۷۰ الهام گرفته شده است. این یک رویکرد یادگیری بدون نظارت را دنبال میکند و شبکه خود را از طریق یک الگوریتم یادگیری رقابتی آموزش می دهد SOM .برای تکنیکهای خوشه بندی و نقشه برداری)یا کاهش ابعاد(برای نگاشت داده های چند بعدی بر رو ی ابعاد پایینتر استفاده میشود که به افراد اجازه م یدهد مشکلات پیچیده را برای تفسیر آسان کاهش دهند SOM .دارای دو لایه است، یکی لایه ورودی و دیگری لایه خروجی است.

توضیحات بیش تر :

برخی بینش در مورد نقشه های خودسازماندهی مقاله اصلی منتشر شده توسط Teuvo Kohonen در سال ۱۹۹۸۱ شامل شرح مختصری و استادانه از این تکنیک است. در آنجا، توضیح داده شده است که یک نقشه خودسازماندهی به عنوان یک شبکه (معمولاً دو بعدی) از گره ها، الهام گرفته از یک شبکه عصبی توصیف می شود. ارتباط نزدیک با نقشه، ایده مدل است، یعنی مشاهده دنیای واقعی که نقشه سعی در نشان دادن آن دارد. هدف این تکنیک نمایش مدل با تعداد ابعاد کمتر، در حالی که روابط شباهت گره های موجود در آن را حفظ می کند.

برای به دست آوردن این شباهت، گره‌ها در نقشه از نظر فضایی سازمان‌دهی می‌شوند تا هر چه بیشتر به یکدیگر شباهت داشته باشند، نزدیک‌تر باشند. به همین دلیل، SOM یک راه عالی برای تجسم الگو و سازماندهی داده ها است. برای به دست آوردن این ساختار، نقشه از یک عملیات رگرسیون برای تغییر موقعیت گره ها به منظور به روز رسانی گره ها، یک عنصر از مدل استفاده می شود.

) در یک زمان. عبارتی که برای رگرسیون استفاده می شود:

این بدان معناست که موقعیت گره nبا اضافه کردن فاصله از آن به عنصر داده شده، ضرب در ضریب همسایگی نورون برنده، به روز می شود.

. برنده یک عنصر، گره شباهت بیشتری در نقشه به آن است، که معمولاً توسط گره نزدیکتر با استفاده از فاصله اقلیدسی اندازه گیری می شود (اگرچه در صورت لزوم می توان از معیار مشابهت متفاوتی استفاده کرد).

از طرف دیگر، محله به عنوان یک هسته کانولوشن مانند برای نقشه اطراف برنده تعریف شده است. با انجام این کار، می‌توانیم برنده و نورون‌های نزدیک‌تر به عنصر را به‌روزرسانی کنیم و نتیجه نرم و متناسبی به‌دست آوریم. تابع معمولاً به عنوان یک توزیع گاوسی تعریف می‌شود، اما سایر پیاده‌سازی‌ها نیز چنین هستند. یکی از مواردی

که قابل ذکر است یک همسایگی حبابی است که نورون هایی را که در شعاع برنده قرار دارند (بر اساس تابع دلتای مجزای کرونکر) به روز می کند که ساده ترین تابع همسایگی ممکن است.

اصلاح تکنیک

برای استفاده از شبکه برای حل TSP، مفهوم اصلی درک این است که چگونه تابع همسایگی را تغییر دهیم. اگر به جای یک شبکه، یک آرایه دایره ای از نورون ها را اعلام کنیم، هر گره فقط از نورون های جلو و پشت خود آگاه خواهد بود. یعنی شباهت درونی فقط در یک بعد کار خواهد کرد. با انجام این اصلاح جزئی، نقشه خودسازماندهی مانند یک حلقه الاستیک رفتار می‌کند و به شهرها نزدیک‌تر می‌شود اما به لطف عملکرد محله تلاش می‌کند تا محیط آن را به حداقل برساند.

اگرچه این اصلاح ایده اصلی پشت این تکنیک است، اما آنطور که هست کار نخواهد کرد: الگوریتم به سختی در هیچ یک از زمان ها همگرا می شود. برای اطمینان از همگرایی آن، می‌توانیم نرخ یادگیری را در نظر بگیریم،

، برای کنترل اکتشاف و بهره برداری از الگوریتم. برای به دست آوردن اکتشاف بالا در ابتدا، و بهره برداری بالا پس از آن در اجرا، ما باید هم در تابع همسایگی و هم در نرخ یادگیری یک کاهش را لحاظ کنیم. کاهش سرعت یادگیری، جابجایی تهاجمی کمتری از نورون‌های اطراف مدل را تضمین می‌کند، و پوسیدگی همسایگی منجر به بهره‌برداری متوسط‌تر از حداقل محلی هر بخش از مدل می‌شود. سپس، رگرسیون ما را می توان به صورت زیر بیان کرد:

$$n_{t+1} = n_t + \alpha_t \cdot g(w_e, h_t) \cdot \Delta(e, n_t)$$

جایی که نرخ یادگیری در یک زمان معین است، و (g) تابع گاوسی است که در یک برنده و با پراکندگی همسایگی از

. تابع زوال شامل ضرب کردن دو تخفیف داده شده است،

، برای میزان یادگیری و فاصله محله.

$$\alpha_{t+1} = \gamma_\alpha \cdot \alpha_t, \quad h_{t+1} = \gamma_h \cdot h_t$$

این عبارت در واقع کاملاً شبیه به Q-Learning است و همگرایی به روشی مشابه با این تکنیک جستجو می شود. تحلیل رفتن پارامترها می تواند در کارهای یادگیری بدون نظارت مانند موارد فوق مفید باشد. همچنین شبیه به عملکرد تکنیک Quantization بردار یادگیری است که توسط Teuvo Kohonen توسعه یافته است.

در نهایت، برای به دست آوردن مسیر از SOM، فقط لازم است یک شهر را با نورون برنده اش مرتبط کنیم، حلقه را از هر نقطه شروع کنیم و شهرها را بر اساس ترتیب ظاهر نورون برنده آنها در حلقه مرتب کنیم. اگر چندین شهر به یک نورون نگاشت می شوند، به این دلیل است که ترتیب عبور از چنین شهرهایی توسط SOM در نظر گرفته نشده است (به دلیل عدم ارتباط با فاصله نهایی یا به دلیل عدم دقت کافی). در آن صورت می توان هرگونه سفارش احتمالی را برای چنین شهرهایی در نظر گرفت.

با استفاده از دیتاست فرضی به حل مسئله می پردازیم:

ابتدا از روی فایل داده شده مختصات شهر ها را استخراج کرده و در آرایه ای ذخیره و نرمالایز میکنیم.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import csv



file = open('Cities.csv')
my_file = csv.reader(file)
lines = []
x = []
y = []
for line in my_file:
    x_point = float(line[0].split()[1])
    y_point = float(line[0].split()[2])
    x.append(x_point)
    y.append(y_point)
    lines.append([x_point, y_point])

x_y = np.array(lines)
c = x_y.shape[0]
ratio = np.sqrt((max(x) - min(x)) * (max(x) - min(x)) + (max(y) * min(y)) * (max(y) * min(y)))
normalized_x_y = (x_y - np.array([min(x), min(y)])) / ratio
```

سپس چهار تابع کمکی، به ترتیب برای ساخت شبکه، احتساب مختصات نورون برنده، احتساب مقادیر همسایگی و پالت کردن نتایج تعریف میکنیم.

```python
# Generate a neuron network of a given size
def som_network(size):
    return np.random.rand(size, 2)

def closest_neuron(network, city):
    dist = network - city
    dist = dist ** 2
    dist = np.sum(dist, axis=1)
    return np.where(dist == np.amin(dist))

# Get the range gaussian of given radix around a center index.
def get_neighborhood(center, r, domain):
    if r < 1:
      r = 1
    deltas = np.absolute(center - np.arange(domain))
    distances = np.minimum(deltas, domain - deltas)
    return np.exp(-(distances * distances) / (2 * (r * r)))

# Plot a graphical representation of the problem
def plot_city_network(network, coordinates):
    fig = plt.figure(figsize=(5, 5), frameon = False)
    axis = fig.add_axes([0,0,1,1])
    axis.set_aspect('equal', adjustable='datalim')
    plt.axis('off')
    axis.scatter(coordinates[:, 0], coordinates[:, 1], color='red', s=4)
    axis.plot(network[:,0], network[:,1], 'r.', ls='-', color='#0063ba', markersize=2)
    plt.show()
```

در نهایت این مسئله به همگرایی نهایی نرسیده است اما در صورت بهینه تر کردن کد خواهد رسید

به طور خلاصه :

MLP مناسب نیست اما دو روش دیگر مناسب اند

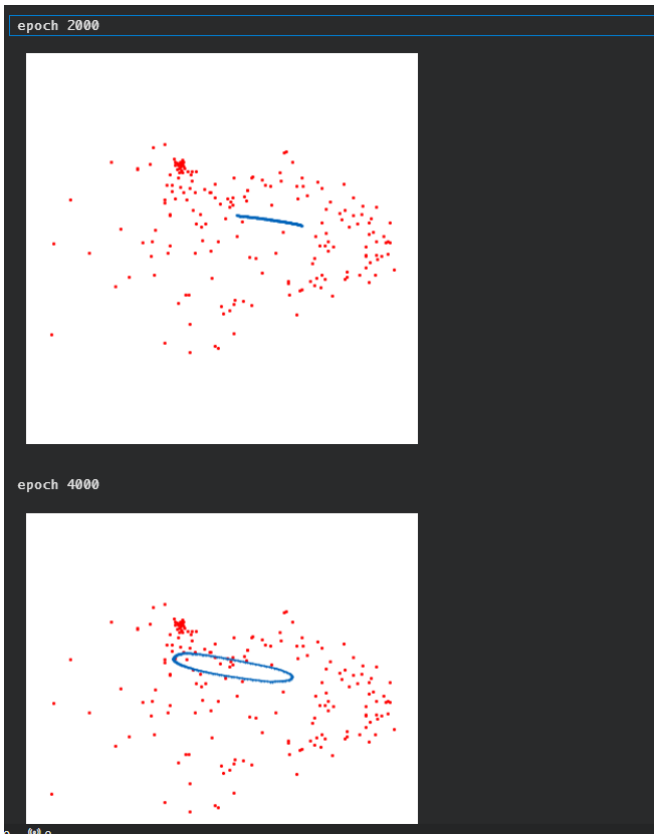
حل این سوال با استفاده از MLP غیر ممکن است. برای استفاده از این روش، باید label برای هر شهر داشته باشیم که چون در اینجا م.ان شهرها در مسیر را نمی دانیم، پس نمی توانیم این مسئله را با این روش حل کنیم

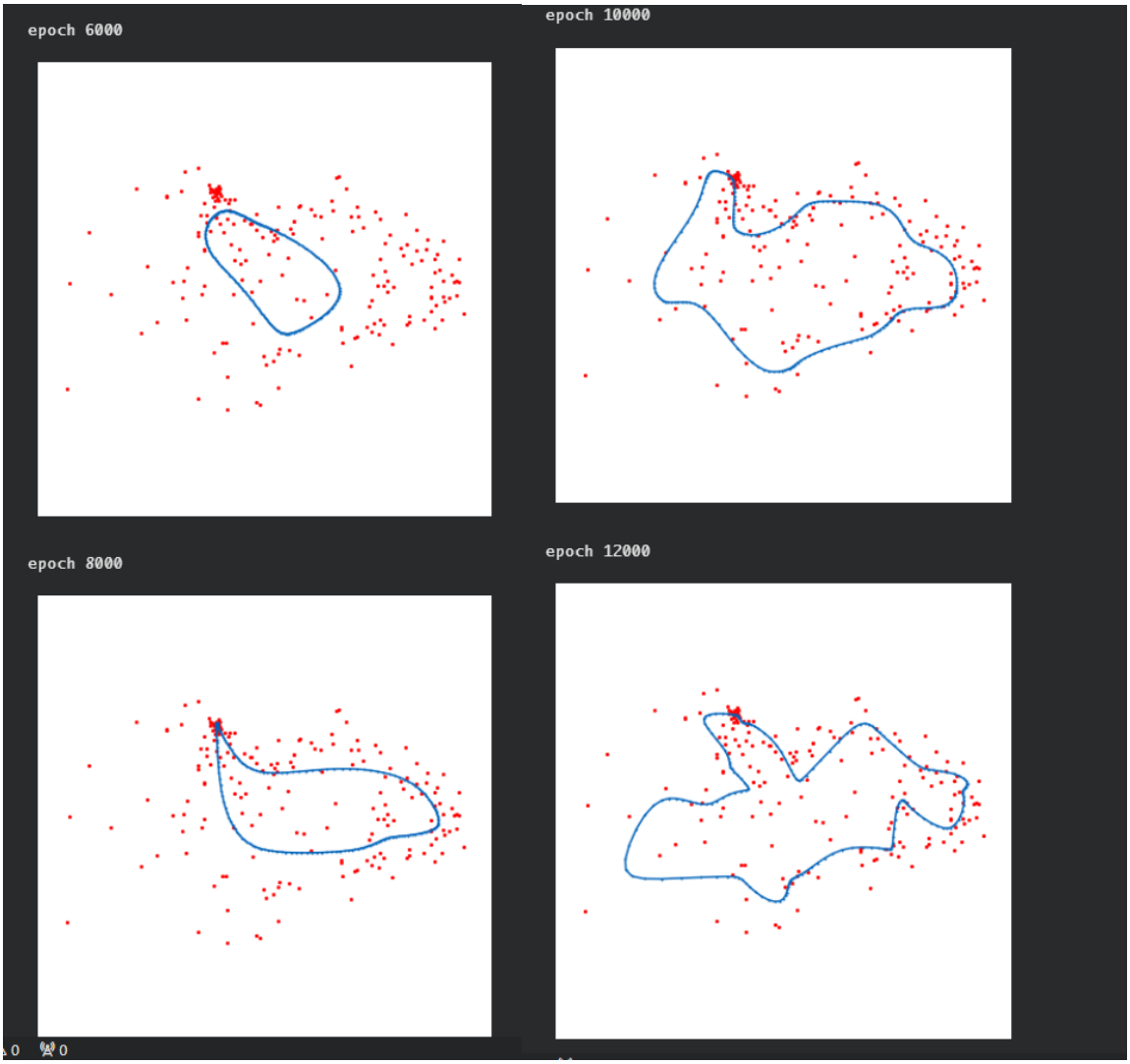
ا این حال، آموزش یک MLP برای TSP می تواند چالش برانگیز باشد، و یافتن یک بهینه استراتژی معماری و آموزش ممکن است به تلاش قابل توجهی نیاز داشته باشد. MLP ها ممکن است مشکل داشته باشند با گرفتن ماهیت ترکیبی TSP، و عملکرد آنها ممکن است متفاوت باشد بر اساس اندازه و پیچیدگی مسئله و شاید بتوان با را حل های ترکیبی از MLP ان ها را حل کرد.

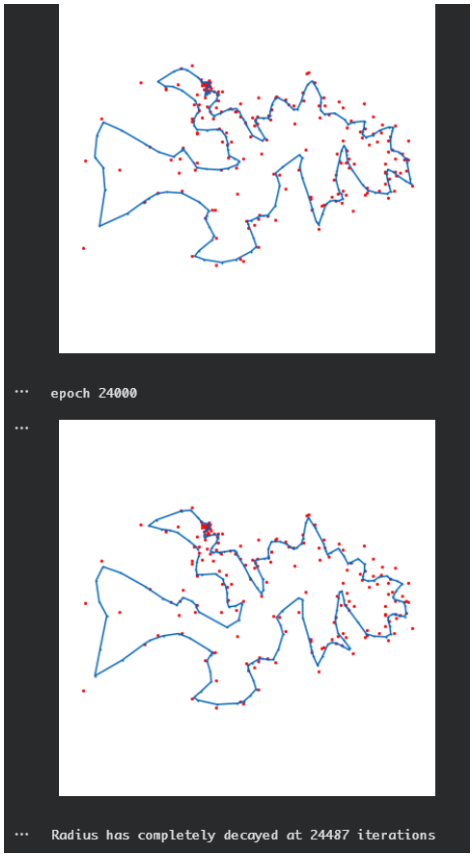
۱. این مسئله با استفاده از Hopfield قابل حل می‌باشد. به این صورت که برای هر شهر، یک نورون در شبکه هاپفیلد در نظر می‌گیریم. همچنین برای مقدار دهی اولیه وزن‌ها نیز از فاصله بین شهرها استفاده می‌کنیم. سپس شبکه خود را train کرده و اگر یک وزن ۱ بشود، جایگاه آن شهر را در مسیر نشان می‌دهد. زمانی که یکی از نورون‌های شبکه در جایگاه یک نورون متفاوت ۱ شود، نشان دهنده رسیدن به مسیر می‌باشد و همچنین آن مسیر، بهترین مسیر می‌شود.

از SOM نیز می‌توانیم برای حل این سوال استفاده کنیم. نخست، یک شبکه با تعداد نورون‌های برابر با تعداد شهرها می‌سازیم. برای بردار وزن هر نورون، ۲ عنصر داریم. ورودی‌های شبکه در اینجا، مختصات شهرهایمان هستند که بهتر است نرمالیزه شوند. وزن‌های شبکه را نیز در ابتدا به صورت تصادفی initialize می‌کنیم. در هر بار، یک نورون انتخاب می‌شود و شبکه شکل می‌گیرد. بنابراین برای هر شهر، یک نورون برنده داریم که مکان آن شهر در مسیر را مشخص می‌کند.

```
"""Solve the TSP using a Self-Organizing Map."""

iterations = 50000
lr = 0.8
# The population size is 8 times the number of cities
radius = c * 8

# Generate an adequate network of neurons
net = som_network(c)

for iteration in range(iterations):
  rand_city = np.random.randint(0, c)
  winner = closest_neuron(net, normalized_x_y[rand_city])[0][0]
  neighbour = get_neighborhood(winner, radius // 10, c)
  net += neighbour[:,np.newaxis] * lr * ( normalized_x_y[rand_city] - net)
  lr = lr * 0.99997
  radius = radius * 0.9997

  # Check for plotting interval
  if (iteration + 1) % 2000 == 0:
    print(f'epoch {iteration + 1}')
    plot_city_network(net, normalized_x_y)

  # Check if any parameter has completely decayed
  if radius < 1:
    print('Radius has completely decayed at {} iterations'.format(iteration))
    break

  if lr < 0.001:
    print('Learning rate has completely decayed at {} iterations'.format(iteration))
    break
```


epoch 2000
epoch 4000


epoch 6000
epoch 10000
epoch 8000
epoch 12000



پاسخ : Bing

Among the three neural networks mentioned, Hopfield networks have the potential to solve the Traveling Salesman Problem (TSP). The other two, SOM (Kohonen) and MLP (multilayer perceptron), are not well-suited for this task due to their inherent limitations.

Hopfield Networks and the TSP

Hopfield networks are a type of recurrent neural network capable of storing and retrieving memories, making them suitable for combinatorial optimization problems like the TSP. In the context of TSP, a Hopfield network can be used to encode the problem as an energy function, where the energy represents the total cost of a particular tour. The network then iteratively updates the state of its neurons to minimize this energy, ultimately converging to a solution that represents a low-cost tour.

Algorithm for TSP using Hopfield Networks

١. Encoding the Problem: Represent each city as a neuron in the Hopfield network. The weights between neurons represent the distances between cities.

٢. Energy Function: Define an energy function that penalizes long distances and rewards visiting all cities exactly once.

٣. Network Update Rule: Update the state of each neuron based on the local energy minimization principle, aiming to reduce the overall energy of the network.

٤. Convergence: The network continues updating until it reaches a stable state, representing a low-cost tour.

Network Structure

The Hopfield network for TSP has a fully connected structure, where each neuron is connected to every other neuron. This allows the network to capture the global relationships between cities and their distances.

Limitations of SOM and MLP

- SOM (Kohonen): SOM is primarily used for unsupervised learning tasks like clustering and dimensionality reduction. It is not well-equipped to handle optimization problems like TSP, which require explicit search strategies.
- MLP (Multilayer Perceptron): MLP is designed for supervised learning tasks, where it learns to map inputs to outputs based on labeled data. While MLP can be used for approximation problems, it struggles with combinatorial optimization problems like TSP.

In conclusion, Hopfield networks offer a promising approach to solving the TSP due to their ability to store and retrieve memories and their ability to minimize energy functions. However, it is important to note that Hopfield networks may not always find the optimal solution, and their performance can be sensitive to the choice of parameters.

ساختار شبکه هاپفیلد:

: N: $number\ of\ neurons$

Θi : $threshold\ of\ neuron\ i$

K: $number\ of\ patterns$

P: $List\ of\ patterns$

P = [$p1$, $p2$, … , pK]

$xi\ k$: $input\ of\ neuron\ i\ in\ pattern\ k\ pk$ = [$x1\ k$, $x2\ k$, … , $xN\ k$]

wi,j : $weight\ between\ neuron\ i\ to\ j\ determined\ by\ Hebbian\ rule$

$wi,j = \sum xi\ k\ K\ k$=1 $xj\ k\ wi,i$ = 0 $wi,j = wj,i$

$u(i,t + 1)$:$linear\ activation\ of\ neuron\ i\ at\ time\ t$ + 1

$u(i,t + 1) = \sum wi,j\ N\ j$=1 $a(j,t) - \Theta i$

$a(i,t + 1)$: $sign\ activation\ of\ neuron\ i\ at\ time\ t$ + 1

$a(i,t + 1) = sign(u(i,t + 1))$

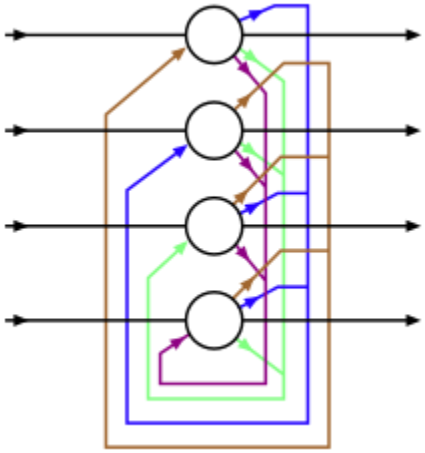
$E(i,t)$:$Energy\ of\ neuron\ i\ at\ time\ t$

$E(i,t) = -a(i,t)u(i,t) = -a(i,t)(\sum wi,j\ N\ j$=1 $a(j,t) - \Theta i)$

$E(t)$: $Total\ Energy$

$E(t) = \sum E(i,t)\ N\ i$=1 $= -\sum a(i,t)(\sum wi,j\ N\ j$=1 $a(j,t) - \Theta i)\ N\ i$=1 $= -\sum\sum wi,ja(i,t)a(j,t)$
$N\ j$=1 $N\ i$=1 $+\sum a(i,t)\Theta i\ N\ i$=1

مراحل الگوریتم: ابتدا با توجه به لیست پترن ها و قاعده هبیان ماتریس وزن را محاسبه می کنیم. سپس برای بررسی پایدار بودن ورودی جدید طی چند مرحله(Epochs) مقادیر a و انرژی را به دست میآوریم. این کار را تا زمانی ادامه میدهیم ک ه یکی از سه حالت زیر رخ دهد 1. :دو a آخر و پشت سر هم یکسان باشند. در این صورت در میان لیست a های ذخیره شدهa یی که کمترین انرژی را دارد خروجی می دهیم 2. .در لحظه اول aبه دست آمده با ورودی برابر باشد. در این صورت ورودی پایدار بوده و خروجی نیز است 3. .در صورتی که اتفاقات باال نیفتد بایستی تمام Epoch ها را کامل برویم و پس از پایان از لیست a هاa یی که کمترین انرژی را دارد برگردانیم ✓ .در سواالت تمرین از اثر Threshold صرف نظر شده و مقدار آن برابر ۰ میباشد ما هم در محاسبات خود آن را صفر در نظر می گیریم.



Step 1 – Initialize the weights, which are obtained from training algorithm by using Hebbian principle.

Step 2 – Perform steps 3-9, if the activations of the network is not consolidated.

Step 3 – For each input vector **X**, perform steps 4-8.

Step 4 – Make initial activation of the network equal to the external input vector **X** as follows –

$$y_i = x_i \quad for\, i = 1\, to\, n$$

Step 5 – For each unit $\mathbf{Y_i}$, perform steps 6-9.

Step 6 – Calculate the net input of the network as follows –

$$y_{ini} = x_i + \sum_j y_j w_{ji}$$

Step 7 – Apply the activation as follows over the net input to calculate the output –

$$y_i = \begin{cases} 1 & if\, y_{ini} > \theta_i \\ y_i & if\, y_{ini} = \theta_i \\ 0 & if\, y_{ini} < \theta_i \end{cases}$$

Here θ_i is the threshold.

Step 8 – Broadcast this output $\mathbf{y_i}$ to all other units.

Step 9 – Test the network for conjunction.

Energy Function Evaluation

An energy function is defined as a function that is bonded and non-increasing function of the state of the system.

Energy function **E_f**, also called **Lyapunov function** determines the stability of discrete Hopfield network, and is characterized as follows –

$$E_f = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j w_{ij} - \sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n} \theta_i y_i$$

Condition – In a stable network, whenever the state of node changes, the above energy function will decrease.

Suppose when node **i** has changed state from $y_i^{(k)}$ to $y_i^{(k+1)}$ then the Energy change ΔE_f is given by the following relation

$$\Delta E_f = E_f(y_i^{(k+1)}) - E_f(y_i^{(k)})$$

$$= -\left(\sum_{j=1}^{n} w_{ij} y_i^{(k)} + x_i - \theta_i\right)(y_i^{(k+1)} - y_i^{(k)})$$

$$= -(net_i)\Delta y_i$$

Here $\Delta y_i = y_i^{(k+1)} - y_i^{(k)}$

The change in energy depends on the fact that only one unit can update its activation at a time.