

به نام خدا



درس هوش محاسباتی

تمرین دوم

مدرس : دکتر مزینی

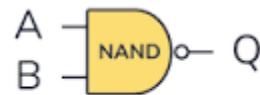
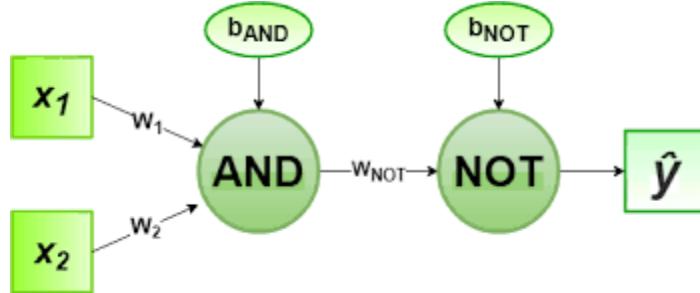
دستیاران : بیان دیوان آذر - محمود کلانتری

دانشجو : سارا سادات یونسی / ۹۸۵۳۳۰۵۳

فهرست

سوال ۱	صفحه ۳
سوال ۲	صفحه ۷
سوال ۳	صفحه ۱۹
سوال ۴	صفحه ۲۹
سوال ۵	صفحه ۳۴
سوال ۶	صفحه ۴۵
منابع استفاده شده	صفحه ۵۷

۱- با استفاده از Neuron Adaline تابع گیت NAND را مدلسازی کنید. برای حل مسئله مقدار نرخ یادگیری (α) را برابر با ۰،۱، وزنهای اولیه را به صورت رندوم در نظر بگیرد و مدل را برای ۴ مرحله آموزش دهید و تمام مراحل آموزش را یاداشت کنید



A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Neuron Adaline

ابتدا توضیحی درباره ای این روش می پردازم :

از عملکرد فعال سازی دوقطبی استفاده می کند. نورون آadalین را می توان با استفاده از قانون دلتا یا قانون حداقل میانگین مربع (LMS) یا قانون widrow-hoff آموزش داد. ورودی خالص با مقدار هدف مقایسه می شود تا سیگنال خطا محاسبه شود. بر اساس الگوریتم تمرین تطبیقی وزن ها تنظیم می شوند ساختار اصلی آadalین شبیه پرسپترون است که دارای یک حلقه بازخورد اضافی است که با کمک آن خروجی واقعی با خروجی مورد نظر/هدف مقایسه می شود. پس از مقایسه بر اساس الگوریتم تمرین، وزن ها و سوگیری ها به روز می شوند.

مرحله ۰: وزن ها را مقداردهی کنید و بایاس ها روی مقادیر تصادفی تنظیم می شوند اما روی صفر نیستند، همچنین نرخ یادگیری α را مقداردهی کنید.

مرحله ۱ : مراحل ۷-۲ را زمانی که شرط توقف نادرست است انجام دهید

مرحله ۲ : مراحل ۳-۵ را برای هر جفت تمرین دوقطبی t : y_{in} انجام دهید.

مرحله ۳ : هر واحد ورودی را به صورت زیر فعال کنید .

$$x_i = s_i (i=1 \text{ to } n)$$

مرحله ۴ : ورودی خالص را بدست آورید

$$y_{in} = \sum i x_i w_i + b$$

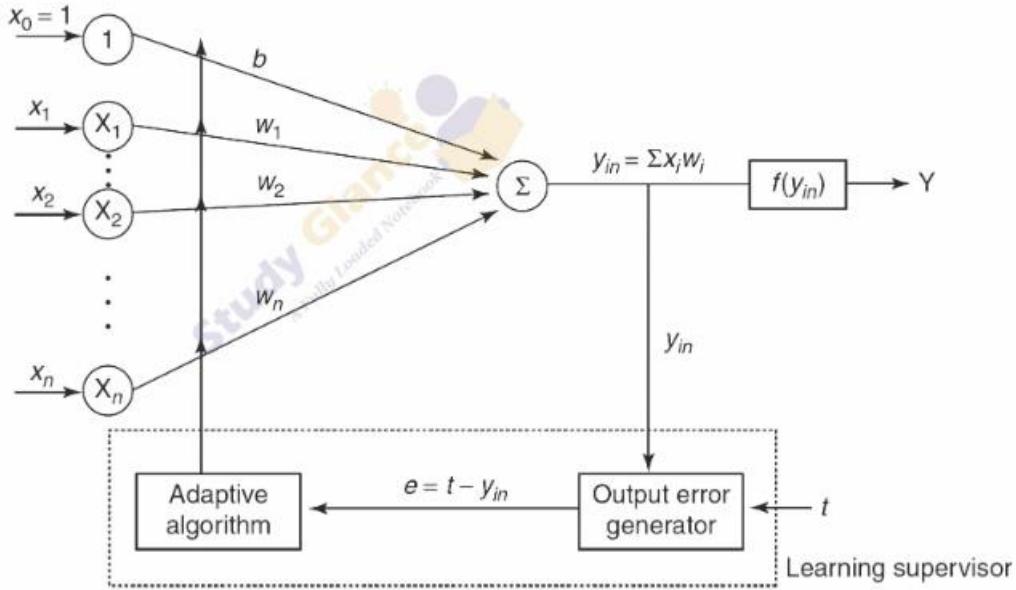
مرحله ۵ : تا زمانی که حداقل میانگین مربع $(t - y_{in})$ به دست آید، وزن و بایاس را به صورت زیر تنظیم کنید.

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in}) x_i$$

$$b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$$

$$E = (t - y_{in})^2$$

مرحله ۷ - شرایط توقف را آزمایش کنید، اگر خطای ایجاد شده کمتر یا برابر با تلورانس مشخص شده باشد، متوقف شوید.



کد اجرا شده و توضیح آن :

از مشتق دومتابع خطاب نسبت به وزن، نرخ یادگیری را برای هر وزن به طور جداگانه تنظیم می کند. این کد شامل مراحل زیر است:

- وارد کردن کتابخانه های `matplotlib.pyplot` و `numpy` برای کار با آرایه ها و نمایش نمودارها
- تعریف جفت های ورودی- خروجی دروازه NAND در دو آرایه `X` و `y`
- تعریف معماری شبکه عصبی Adaline که شامل اندازه لایه ورودی و خروجی، وزن ها و بایاس است
- تعریف تابع فعال سازی که در این مثال تابع هویساید است که ۱ را برای ورودی های مثبت و ۰ را برای ورودی های منفی برمی گرداند
- تعریف پیش خور (forward propagation) که شامل محاسبه خروجی شبکه برای هر ورودی با استفاده از تابع جمع وزن دار و اعمال تابع فعال سازی است
- تعریف پس خور (backward propagation) که شامل به روز رسانی وزن ها و بایاس با استفاده از قاعده یادگیری Adaline است
- آموزش شبکه عصبی Adaline در ۴ دوره (epoch) با استفاده از حلقه های `for`

```

import numpy as np
import matplotlib.pyplot as plt
# Define the NAND gate input-output pairs
x = np.array([[1, 1], [1, -1], [-1, 1], [-1, -1]])
y = np.array([-1, 1, 1, 1])
# print(y.type)

print("lable_1 : {},lable_2 : {},lable_3 : {},lable_4: {}".format(y[0],y[1],y[2],y[3] ))
print("")
print("")

# Define the Adaline neuron architecture
input_layer_size = 2
output_layer_size = 1

# Initialize the weights and bias randomly (Bias=0.1)
w1 = np.random.uniform(low=0, high=1, size=(1,))
w2 = np.random.uniform(low=0, high=1, size=(1,))
bias = 0.1

# Define the activation function
def activation(x):
    return 1 if x >= 0 else 0

# Define the forward propagation
# Train the Adaline neuron in 4 epochs

```

چون تابع ما nand است ابتدا یک دیتابست مطابق با لاجیک مان درست می کنیم سپس فقط ۱ و ۰ است که به ما صفر می دهد و با بقیه متفاوت است حالا به ان ها لیبل می دهیم و وزن های رندوم و بایاس را اضافه می کنیم و نرخ یادگیری را نیز ۰.۱ می کذاریم همچنین یک تابع فعالسازی هم پیاده سازی کردیم بعد از طی کردن مراحل پس خور و پیش خور و ۴ مرحله یادگیری مشاهده می کنیم که اولین لیبل موزد نظر ما مقداری منفی و بقیه آن ها مقداری مثبت است.

```

for epoch in range(4):
    for i in range(len(x)):
        data = x[i]
        # z = sigma(w1 * xi) + w0
        yy = w1 * data[0] + w2 * data[1] + bias
        y_predicted=activation(yy)

        # Define the backward propagation
        learning_rate = 0.1
        w1 += learning_rate * (y[i] - y_predicted) * data[0]
        w2 += learning_rate * (y[i] - y_predicted) * data[1]
        bias += learning_rate * (y[i] - y_predicted)

    # Test the Adaline neuron on new inputs
    for i in range(len(x)):
        y_predicted = w1 * x[i][0] + w2 * x[i][1] + bias
        print("predicted was: {}, actual label was: {}".format(y_predicted[0], y[i]))

```

[7]: ✓ 0.0s

• lable_1 : -1,lable_2 : 1,lable_3 : 1,lable_4: 1

predicted was: -0.5262771846909429, actual label was: -1
predicted was: 0.1418793428647928, actual label was: 1
predicted was: 0.2581206571352072, actual label was: 1
predicted was: 0.926277184690943, actual label was: 1

return $i : y = b + \sum_{j \neq i} w_j x_j$

$$\text{Rule: } b(\text{new}) = b(\text{old}) + \alpha(\text{dsgn})$$

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(\text{dsgn}) x_i$$

$$\alpha = 0.1 \quad \text{bias} = y_1$$

ساده نسبت می سازد و بروزگار است: weight

w_1	w_2	b	total
1	1	1	-1
1	-1	1	1
-1	1	1	1
-1	-1	1	1

$$\begin{array}{l} \textcircled{1} \text{ input } 1,1 \rightarrow w_1 = 0.1 \quad w_2 = 0.1 \\ \text{output } -1 \end{array} \quad y = 0.1 + 0.1 \alpha_1 + 0.1 \alpha_2 = 0.2$$

$$\begin{array}{l} w_1 = 0.1 + 0.1(-1-0.1) = 0.09 \\ w_2 = 0.1 + 0.1(-1+0.1) = 0.08 \\ b = 0.1 + 0.1(-1+0.1) = -0.1 \end{array}$$

پنجمین مرحله

$$\textcircled{2} \quad w_1 = 0.09, w_2 = 0.08 \quad \therefore \text{با توجه به این دو، باید مجموعه داده را برای آنها تغیر داد}$$

$$y_1 = 0.09 \alpha_1 + 0.08 \alpha_2 - 1 = -0.1$$

$$\begin{array}{l} \text{Input: } 1, -1 \\ \text{Output: } 1 \end{array}$$

$$b = 0.09 + 0.1(1 - (-0.1)) = 0.1$$

$$w_1 = 0.09 + 0.1(1 + 0.1) \alpha_1 = 0.1 \quad w_2 = 0.08 + 0.1(1 + 0.1) \alpha_2 = 0.09$$

$$y = 0.1 \alpha_1 + 0.09 \alpha_2 - 0.1 = 0.0$$

پنجمین مرحله

$$\textcircled{3} \quad \text{input: } (-1, 1) \quad \text{update } w_1 = 0.1 \alpha_1 + 0.1(1 + 0.1) = 0.1$$

$$\text{output: } 1 \quad \text{update } w_2 = 0.09 + 0.1(1 + 0.1) = 0.1$$

$$w_1 = 0.1$$

$$w_2 = 0.1$$

$$b = 0.1$$

$$\text{update } b = 0.1(1 - 0.1) = 0.09$$

$$\begin{array}{l} \text{پنجمین مرحله} \\ + 0.1 = 0.1 \end{array}$$

$$y = 0.1 \alpha_1 + 0.1(1 - 0.1) = 0.09$$

$$\textcircled{4} \quad \text{input: } (-1, -1) \quad \text{update } w_1 = 0.1 \alpha_1 + 0.1(1 + 0.1) = -0.09$$

$$\text{output: } 1$$

$$w_1 = 0.09$$

$$w_2 = 0.1$$

$$b = 0.14$$

$$\text{update } w_2 = 0.1 \alpha_2 + 0.1(1 + 0.1) = 0.1$$

پنجمین مرحله

$$\text{update } b = 0.1(1 - 0.1) + 0.1 = 0.1$$

بحث در مورد این آندرودی می‌شود از این سری برای این آندرودی می‌شود و مدارس علمی
 مارچ به مرکز اسناد ملی موارد نهاد
 لوح برآورده از مرکز مارس کرد.
 ۱۵-۶
 مارس
 ۲۰۱۴ = ۱۵

۲- به سوالات زیر پاسخ دهید.

(الف) تفاوت تابع فعال سازی خطی و غیر خطی چیست؟

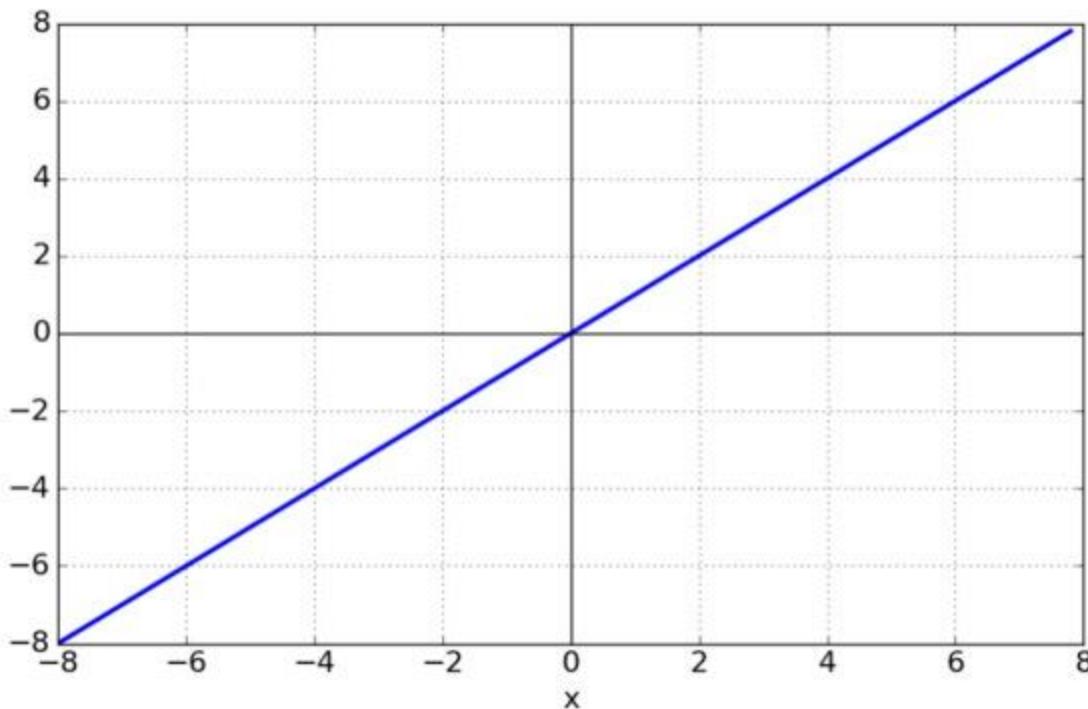
پاسخ (الف) تابع فعال سازی یک تابع ریاضی است که در هر نورون یک شبکه عصبی اعمال می‌شود و خروجی آن را به لایه بعدی منتقل می‌کند. تابع فعال سازی می‌تواند خطی یا غیرخطی باشد. تابع فعال سازی خطی یک تابع است که خروجی آن به صورت مستقیم به ورودی آن بستگی دارد. به عبارت دیگر، تابع فعال سازی خطی یک تابع است که نمودار آن یک خط راست است. مثلاً تابع هویت (identity function) که $f(x) = x$ را بر می‌گرداند، یک تابع فعال سازی خطی است.

تابع فعال سازی غیرخطی یک تابع است که خروجی آن به صورت غیرمستقیم و پیچیده‌تر به ورودی آن بستگی دارد. به عبارت دیگر، تابع فعال سازی غیرخطی یک تابع است که نمودار آن چندین خم و پلک دارد. مثلاً تابع سیگموید (sigmoid function) که $f(x) = \frac{1}{1 + e^{-x}}$ را بر می‌گرداند، یک تابع فعال سازی غیرخطی است.

تفاوت اصلی بین تابع فعال سازی خطی و غیرخطی در این است که تابع فعال سازی خطی قادر به انجام عملهای پایه‌ای و ساده است، اما تابع فعال سازی غیرخطی قادر به انجام عملهای پژوهشگرایانه و پوشش‌دهنده‌تر است. به عبارت دیگر، شبکه‌های عصبی با استفاده از توابع فعال‌سازی غیرخطی می‌توانند الگوهای پیچیده‌تری را یاد بگیرند و پیش‌بینی قابل‌قبولی را در خروجی ارائه کنند.

تابع فعال‌سازی خطی

به تابع فعال‌سازی خطی، «تابع همانی» (Identify Function) «نیز گفته می‌شود. این تابع، بر روی مجموع وزن‌دار ورودی، محاسبات انجام نمی‌دهد و این مقدار را بدون هیچ تغییری به لایه بعد منتقل می‌کند. خروجی این نوع تابع فعال‌سازی در شبکه‌های عصبی، به صورت خطی است و نمی‌توان مقدار آن را در بازه‌ای مشخص تعیین کرد. بنابراین از این نوع تابع فقط در مدل‌های رگرسیون خطی استفاده می‌شود.



می توان تابع ریاضی این فعالساز را به صورت زیر نوشت:

$$f(x)=x$$

تابع فعالسازی خطی، محدودیت‌ها و مشکلات اساسی دارد که در زیر فهرست شده‌اند:

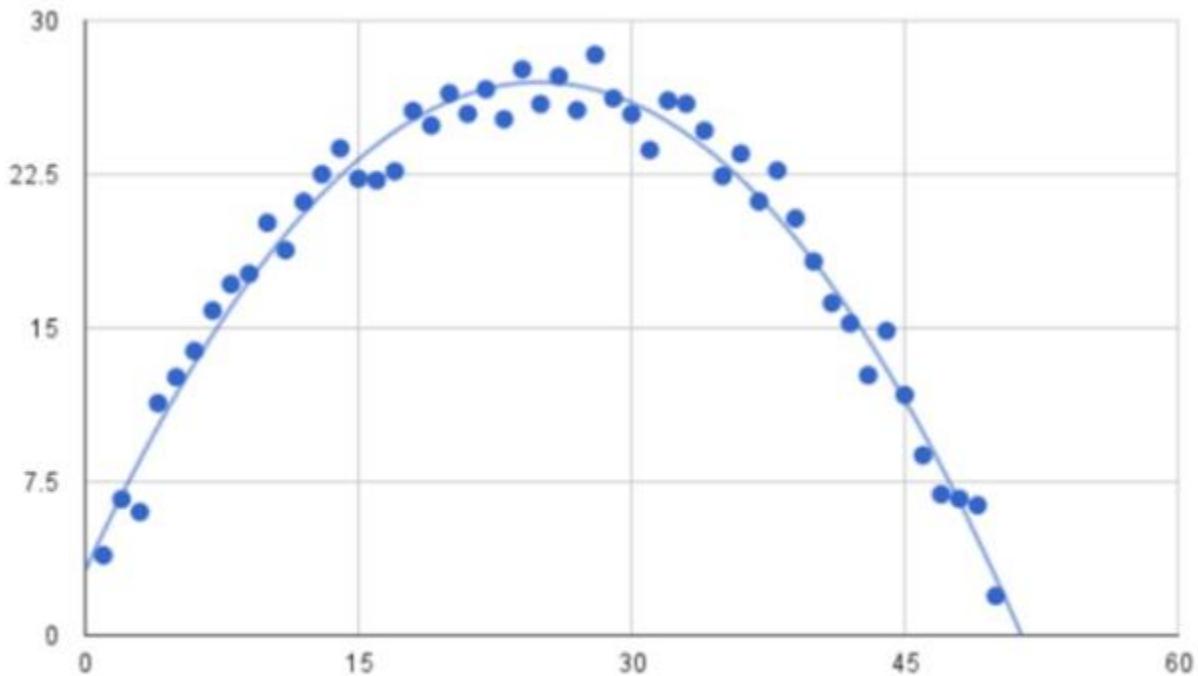
از این تابع نمی‌توان در مرحله انتشار به عقب استفاده کرد زیرا مشتق این تابع برابر با یک عدد ثابت است و هیچ رابطه‌ای با مقدار ورودی X ندارد.

نتیجه چندین تابع خطی بروی یک مقدار ثابت ورودی، یکسان است. بنابراین، اهمیتی ندارد شبکه عصبی از چندین لایه پنهان ساخته شده باشد؛ زیرا خروجی تابع فعالسازی در لایه آخر شبکه عصبی برابر با خروجی تابع فعالسازی در لایه اول است.

زمانی که از پارامترهای زیاد و پیچیده در شبکه عصبی استفاده شده باشد، این نوع تابع فعالسازی عملکرد خوبی نخواهد داشت.

تابع فعالسازی غیرخطی

این نوع توابع بیشترین استفاده را در شبکه‌های عصبی دارند. تعمیم‌پذیری و تطبیق‌پذیری مدل با انواع مختلف داده با استفاده از توابع فعالسازی غیرخطی آسان می‌شود.



استفاده از توابع فعالسازی غیرخطی در شبکه های عصبی گام اضافه تری را به محاسبات هر لایه در مرحله انتشار پیش خور اضافه می کند؛ اما این گام اضافه تر مزیت مهمی دارد. اگر یک شبکه عصبی بدون تابع فعالسازی وجود داشته باشد، هر گره صرفاً بروی ورودی های خود با استفاده از مقادیر وزن ها و بایاس محاسبات خطی انجام می دهد. در این حالت، تعداد لایه های پنهان در شبکه عصبی اهمیت ندارند و تمامی لایه ها عملکردی مشابه دارند زیرا حاصل ترکیب دو تابع خطی، یک تابع خطی خواهد بود. با این که در این حالت، شبکه عصبی ساده تر است، اما شبکه قادر به انجام وظیفه های پیچیده تر نخواهد بود و کاربرد شبکه عصبی فقط به مسائل رگرسیون خطی محدود می شود.

تابع فعالسازی غیرخطی محدودیت های تابع های فعالسازی خطی را رفع کرده اند. ویژگی های اصلی این توابع فعالسازی عبارت اند از:

این تابع مشکل مربوط به مرحله انتشار به عقب را حل کردند. به عبارتی، تابع غیرخطی در مرحله انتشار رو به عقب می تواند مشخص کنند کدام وزن گره ورودی به تشخیص نهایی مدل می تواند کمک بهتری کند. با استفاده از این تابع می توان مسائل مربوط به خروجی های چند گانه را حل کرد. به عبارتی، خروجی تابع غیرخطی، ترکیب غیرخطی از ورودی ها است که از لایه های متعدد عبور کرده است.

تابع فعالسازی غیرخطی در شبکه های عصبی را می توان به چندین نوع تقسیم کرد که در ادامه فهرست شده اند

(Sigmoid | Logistic) تابع فعالسازی سیگموئید

(Tanh | Hyperbolic Tangent) تابع فعالسازی تابع تانژانت هذلولوی

(RELU) تابع فعالسازی یکسوساز

(Leaky RELU) تابع فعالسازی یکسوساز رخنهدار

(Parametric RELU) تابع فعالسازی یکسوساز پارامتریک

(Exponential Linear Units | ELU) تابع فعالسازی واحد های خطی نمایی

تابع فعالسازی Softmax

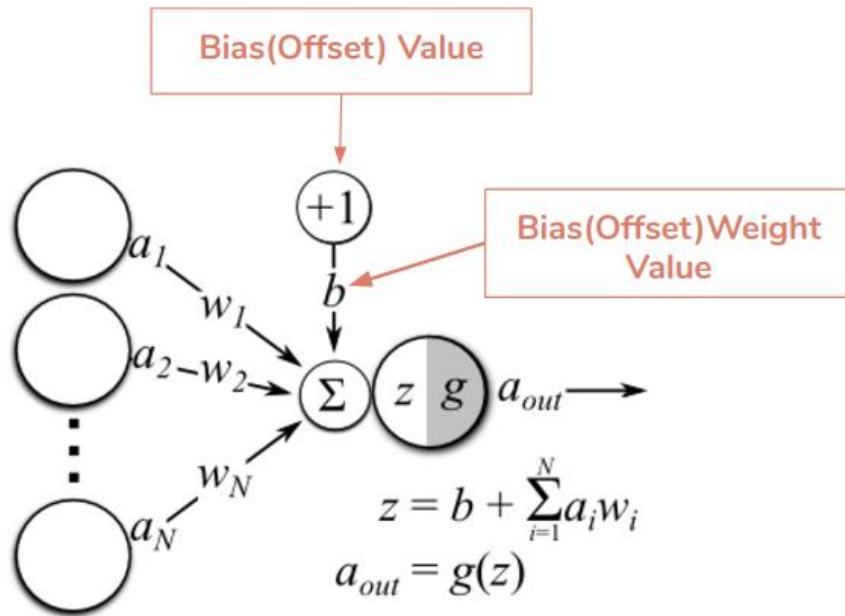
تابع فعالسازی Swish

(Gaussian Error Linear Unit | GELUS)

(Scaled Exponential Linear Unit | SELUS)

ب) اگر در مدل MLP مقادیر اولیه وزن ها و بایوس به حالت زیر در نظر بگیریم ، توضیح دهید آموزش مدل به چه صورت پیش می رود؟

- بایاس رندوم و وزن ها صفر
پاسخ)



باید توجه داشت که این حالت چندین مشکل را به همراه دارد. اول اینکه، اگر وزن ها صفر باشند، خروجی هر نورون صفر خواهد بود و تابع فعال سازی هم تاثیر نخواهد داشت. دوم اینکه، اگر وزن ها صفر باشند، گرادیان خطای نسبت به وزن ها صفر خواهد بود و در نتیجه بهینه سازی نخواهد کار کرد. سوم اینکه، اگر بایاس ها random باشند، خطای پیش بینی شده با خطای واقعی تفاوت زیاد دارد و در نتیجه آموزش طولانی تر خواهد شد.

بنابراین، آموزش مدل MLP با مقادیر اولیه داده شده به صورت زیر پیش می رود:

- در forward propagation، خروجی هر نورون صفر است و فقط بایاس ها تاثیر دارند.
- در backward propagation، گرادیان خطای نسبت به وزن ها صفر است و فقط بایاس ها به روز می شوند.
- در نتیجه، آموزش نامناسب است.

در نهایت مدل نمی تواند آموزش ببیندو هر آنچه در اینپوت های ما بوده است جمع می شود در نتیجه خروجی هر نورون ما صفر می شود و با بایاس ها جمع می شود.

اگر در مدل MLP مقدار اولیه وزن ها صفر باشد و مقدار بایاس رندوم باشد، در ابتدا خروجی هر لایه صفر خواهد بود و همین باعث می شود که خطای بالایی برای تمام داده های آموزشی محاسبه شود. در نتیجه، در دور از آموزش، تمام وزن ها به یکسان

به روز رسانی می شوند و مدل به یک حالت خاص محدود می شود که نمی تواند به صورت درستی آموزش داده شود. بنابراین، بهتر است وزن ها و بایاس با مقادیر کوچک و رندوم شروع شوند تا مدل بتواند به صورت مناسب آموزش داده شود.

Initializing all the weights with zeros leads the neurons to learn the same features during training.

In fact, any constant initialization scheme will perform very poorly. Consider a *neural network* with two hidden units, and assume we initialize all the biases to 0 and the weights with some constant α . If we forward propagate an input in this network, the output of both hidden units will be $relu(\alpha x_1 + \alpha x_2)$. Thus, both hidden units will have identical influence on the cost, which will lead to identical gradients. Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things.

گر همه وزن ها را با صفر مقداردهی کنید، هر *hidden neurons* مستقل از ورودی صفر می شود. بنابراین، وقتی همه نورون های پنهان با وزن های صفر شروع می شوند، همه آنها از شبیه یکسانی پیروی می کنند و به همین دلیل «فقط بر مقیاس بردار وزن تأثیر می گذارد، نه جهت».

(پاسخ)

• **بایاس صفر و وزن ها رندوم**
باید توجه داشت که این حالت چندین نکته را در نظر بگیرد. اول اینکه، اگر بایاس ها صفر باشند، خروجی هر نورون فقط به وزن ها بستگی دارد و تابع فعال سازی کمتر تاثیر دارد. دوم اینکه، اگر وزن ها random باشند، خطای پیش بینی شده با خطای واقعی تفاوت زیاد دارد و در نتیجه آموزش طولانی تر خواهد شد. سوم اینکه، اگر وزن ها random باشند، ممکن است به نقطه چالش برانگیز (saddle point) یا حداقل محلی (local minimum) منجر شوند که باعث کاهش کارایی الگوریتم بهینه سازی می شود بنابراین، آموزش مدل MLP با مقادیر اولیه داده شده به صورت زیر پیش می رود:

- در forward propagation، خروجی هر نورون به صورت random است و فقط وزن ها تاثیر دارند.
- در backward propagation، گرادیان خطای نسبت به وزن ها random است و فقط وزن ها به روز می شوند.
- در نتیجه، آموزش مدل کند است

مدل می تواند آموزش ببیند اما آموزش دادن ان با مشکلاتی رو به رو می شود و کند خواهد بود چون ما وزن هایی بین صفر تا یک به هر نورون اختصاص می دهیم و با وجود نداشتن بایاس مدت زیادی طول خواهد کشید که خروجی های مورد نظر ما تولید شود و دیر همگرا می شود چون مقدار بایاس جمع نمی شود از این رو یا باید یک بایاس مقدار مناسب با ان جمع کنیم یا وزنی بیش از حد معمول به آن اختصاص بدهیم

اگر مقدار اولیه بایاس ها صفر باشد و مقدار وزن رندوم باشد، در ابتدا خروجی هر لایه صفر نخواهد بود ولی به دلیل اینکه وزن ها رندوم هستند، ممکن است خروجی هر لایه به صورت تصادفی تغییر کند. این باعث می شود که مدل به یک حالت خاص محدود نشود و بتواند به صورت مناسب آموزش داده شود. با این حال، بهتر است مقادیر اولیه وزن ها و بایاس ها با مقادیر کوچک و رندوم شروع شوند تا مدل بتواند به صورت بهینه آموزش داده شود.

f a neural network does not have a bias node in a given layer, it will not be able to produce output in the next layer that differs from 0 (on the linear scale, or the value that corresponds to

the transformation of 0 when passed through the activation function) when the feature values are 0.

ج) به نظر شما قابلیت تعمیم در کدام یک از شبکه های عصبی که تاکنون شناخته اید بیشتر و در کدام یک کمتر است؟
پاسخ ج

تفاوت اصلی بین **Perceptron**، **Adaline** و **Perceptrone** گرفته و سپس **error response binary** را محاسبه کرده و از آن برای آپدیت کردن وزن ها، استفاده می کند؛ اما **Adaline** از **response continuous** استفاده کرده و وزن ها را آپدیت می کند. این ویژگی، باعث می شود آپدیت های آن قبل از تعیین شدن **threshold**، بیشتر شبیه ارور واقعی باشد؛ در نتیجه به مدل ما اجازه می دهد سریع تر همگرا شود و در نتیجه قابلیت **generalization** آن بیشتر باشد. همچنین **Adaline**، از **AND** چند **Madaline** به دست می آید و قابلیت حل مسائل غیر خطی را نیز دارد، بنابراین قابلیت تعمیم هم از **Adaline** بیشتر است؛ زیرا **Adaline** فقط برای مسائل تفکیک پذیر خطی قابل کاربرد هست، به علاوه تعداد لایه ها در **Adaline** از **Madaline** بیشتر می باشد. در **MLP**، به دلیل توانایی نگاشت غیر خطی و در نتیجه قابل استفاده بودن آن در مسائل **classification** پیچیده تر، قابلیت تعمیم آن، از همه بیشتر است. بنابراین اگر بخواهیم این مدل ها را براساس قابلیت تعمیم پذیری شان طبقه بندی کنیم، به شرح زیر می باشد

MLP > Madaline > Adaline > Perceptrone

تفاوت اصلی بین این دو، این است که یک **Perceptron** آن پاسخ دودویی (مانند یک نتیجه طبقه بندی) را میگیرد و یک خط را محاسبه میکند که برای آپدیت وزن ها استفاده می شود، در حالی که یک **Adaline** از یک مقدار پاسخ پیوسته برای آپدیت وزنها استفاده می کند. پرسپترون وزن ها را با محاسبه تفاوت بین مقادیر کالس مورد انتظار و پیش بینی شده به روز می کند. به عبارت دیگر، پرسپترون هم یشه + ۱ یا -۱ (مقادیر پیش بینی شده را با +۱ یا -۱ مقادیر مورد انتظار مقایسه میکند. پرسپترون تنها زمانی یاد م میگیرد که خط ایجاد شود. در مقابل، **Adaline** تفاوت بین مقدار کالس مورد انتظار +۱ یا -۱ (و مقدار خروجی **Adaline** پیوسته ع را از تابع خطی محاسبه میکند که میتواند هر عدد واقعی باشد. این بسیار مهم است زیرا به این معنی است که **Adaline** میتواند حتی زمانی که هیچ اشتباهی در طبقه بندی انجام نشده است یاد بگیرد. از آنجایی که **Adaline** هم یشه و پرسپترون فقط پس از خطاهای یاد می گیرد، **Adaline** راه حلی سریعتر از پرسپترون برای همان مشکل پیدا می کند. این واقعیت که **Adaline** این کار را انجام میدهد، اجازه میدهد آپدیتهای آن، قبل از اینکه آستانه تعیین شود، بیشتر شبیه خطای واقعی باشند. که به توبه خود به مدل اجازه میدهد تا سریعتر همگرا شود. پس قابلیت تعمیم پذیری **Perceptron** از **Adaline** بیشتر است. همچنین **Madaline** ترکیب **AND** چند **Adaline** است که بدینهیست نسبت به دو مورد قبلی تعمیمپذیری بیشتری دارد. مزیت **MLP** نیز نسبت به **Perceptron** و **Adaline** کالسیک این است که با اتصال نورونهای مصنوعی در این شبکه از طریق توابع فعالسازی غیرخطی، میتوانیم مرزهای تصمیمگیری پیچیده و غیرخطی ایجاد کنیم که به ما امکان میدهد با مشکالتی که در آن کالسنهای مختلف قابل جداسازی خطی نیستند، مقابله کنیم. پس قابلیت تعمیمپذیری آن از تمام موارد دیگر ذکر شده بیشتر است. (لینک) پس به ترتیب ابتدا **MLP** سپس **Adaline**، **Madaline** و در آخر **Perceptron** است.

شبکه کوهونن شامل یک لایه ورودی و یک لایه خروجی است. لایه خروجی به صورت یک شبکه دو بعدی از نورونها است که هر کدام یک بردار وزن دارند. در طول یادگیری، شبکه کوهونن سعی میکند تا نورونهای خروجی را به نحوی مرتب کند که نماینده‌ی الگوهای موجود در داده‌های ورودی باشند. برای این کار، شبکه کوهونن از دو مرحله رقابت و همکاری استفاده میکند. در مرحله

رقبات، شبکه کوهونن برای هر داده ورودی، نورونی را که بیشترین شباهت را به آن دارد، به عنوان نورون برنده انتخاب میکند. در مرحله همکاری، شبکه کوهونن وزنهای نورون برنده و همسایههای آن را به سمت داده ورودی تغییر میدهد تا شبکه بهتر بتواند دادهها را نمایش دهد.

تعمیم پذیری Kohonen بیشتر از MLP است. الگوریتم Kohonen برای شبکه های خودسازماندهی و تحلیل مؤلفه های اصلی (PCA) استفاده می شود و معمولاً برای تحلیل داده های بدون نظارت و کاهش بعد استفاده می شود. در حالی که MLP برای یادگیری ماشین، شناسایی الگو و پیش بینی استفاده می شود. بنابراین، Kohonen دارای تعییم پذیری بالاتری است زیرا می تواند در مسائل گوناگون از جمله تحلیل تصویر، پردازش زبان طبیعی و شناسایی الگو استفاده شود. در حالی که MLP معمولاً در مسائل پیچیده تر و با داده های بزرگتر استفاده می شود.

There is no definitive answer to whether Kohonen is more generalizable than MLP, as it depends on various factors such as the type and complexity of the problem, the size and quality of the data, the architecture and parameters of the network, and the evaluation criteria. However, some general observations can be made based on the literature:

- Kohonen has the advantage of being able to discover the underlying structure and patterns of the data without requiring any labels or prior knowledge. This can be useful for exploratory data analysis, data visualization, feature extraction, and anomaly detection. Kohonen can also adapt to changing data and learn incrementally. However, Kohonen has the disadvantage of being sensitive to the order and distribution of the data, the choice of the network size and topology, and the initialization and update of the weights. Kohonen may also suffer from overfitting or underfitting if the network is too large or too small for the data.

<https://arxiv.org/abs/2203.12893>

- MLP has the advantage of being able to learn complex nonlinear relationships between the input and output variables and perform various tasks such as classification, regression, prediction, and approximation. MLP can also benefit from various techniques such as regularization, dropout, batch normalization, and optimization algorithms to improve its generalization performance. However, MLP has the disadvantage of requiring a large amount of labeled data and computational resources to train effectively. MLP may also suffer from overfitting or underfitting if the network is too deep or too shallow, or if the learning rate is too high or too low.

<https://link.springer.com/article/10.1007/s11277-022-10079-4>

Therefore, the generalization performance of Kohonen and MLP depends on the specific problem and data at hand, and the optimal network design and configuration. Some studies have compared the generalization ability of Kohonen and MLP on different tasks and domains, and have reported mixed results. For example:

- A study on different deep learning algorithms used in deep neural nets: MLP SOM and DBN <https://link.springer.com/article/10.1007/s11277-022-10079-4>

found that MLP outperformed SOM and DBN on wireless network optimization problems, but SOM performed better on image classification problems.

- A study on generalization aspect of accurate machine learning models for wireless networks <https://link.springer.com/article/10.1007/s12243-021-00853-z>

found that KNN outperformed MLP on wireless network parameter estimation problems, but MLP performed better on wireless network classification problems. Combining the predictions of KNN and MLP NN ensemble did not improve accuracy.

- A study on FAMLP: A Frequency-Aware MLP-Like Architecture For Domain Generalization
<https://arxiv.org/abs/2203.12893>

found that FAMLP, a novel frequency-aware MLP architecture, outperformed the state-of-the-art methods on domain generalization problems, which aim to learn a model that can generalize well across different domains or distributions of data.

ر زمان آموزش دیده
ن عامل بستگی دارد

- تعداد و کیفیت داده های آموزش
- معماری و پارامترهای شبکه عصبی
- روش آموزش و بهینه سازی شبکه عصبی
- مقدار منظم سازی (regularization) و جلوگیری از بیش برازش (overfitting)

به طور کلی، نمی توان گفت که کدام نوع شبکه عصبی قابلیت تعمیم بالاتر یا پایین تری دارد، زیرا این مسئله به مسئله خاص، داده های موجود و روش آموزش بستگی دارد. اما می توان چند نکته را در نظر گرفت:

- شبکه های عصبی با لایه های بسیار زیاد (deep neural networks) عموماً قابلیت ژئال سازی بالاتری نسبت به شبکه های عصبی با لایه های کمتر (shallow neural networks) دارند، زیرا می توانند بازنمایی های پیچیده تر و عمیق تر از داده ها را یاد بگیرند. البته این شرط لازم است که داده های آموزش کافی و مناسب باشند و روش منظم سازی مناسب اعمال شود.
- شبکه های عصبی با لایه های بازگشتی (recurrent neural networks) قابلیت تعمیم خوبی در پردازش داده های دارای توالی (sequence) مانند متن، صوت و ویدئو دارند، زیرا می توانند اطلاعات گذشته را حفظ کنند و به خروجی خود اضافه کنند. البته این نوع شبکه ها نسبت به مشکل ناپدید شدن گرادیان (vanishing gradient) حساس هستند و نسخه های بهبود یافته آمانند (GRU (Gated Recurrent Unit) و LSTM (Long Short-Term Memory) پیدا شده اند.
- شبکه های عصبی با لایه های کانولوشنی (convolutional neural networks) قابلیت تعمیم خوبی در پردازش داده های دارای ساختار شبکه ای (grid) مانند تصویر و صوت دارند، زیرا می توانند ویژگی های محلی و سلسله مراتبی را از داده ها استخراج کنند. البته این نوع شبکه ها نسبت به مشکل بیش برازش (overfitting) حساس هستند و نیاز به روش های منظم سازی مانند batch normalization و dropout دارند

قابلیت تعمیم در شبکه های عصبی پرسپترون چند لایه (MLP) بیشتر است. این شبکه ها قابلیت یادگیری الگوهای پیچیده را دارند و قادر به تعمیم دادن دانش به داده های جدید هستند. همچنین، شبکه های MLP قابلیت تطبیق با مجموعه داده های بزرگ را دارند و به طور کلی به عنوان یکی از قوی ترین شبکه های عصبی برای کاربردهای گوناگون شناخته می شوند. شبکه حافظه کوتاه مدت بلند مدت یا LSTM شاید موفق ترین شبکه RNN باشد زیرا بر مشکلات آموزش یک شبکه تکراری غلبه می کند و به نوبه خود در طیف گسترده ای از برنامه ها استفاده شده است. به طور کلی، مدل های ساده تر کمتر مستعد برازش بیش از حد هستند، زیرا به دلیل سادگی، قوانین تصمیم گیری درشت تر هستند و به تعمیم های بیشتری نیاز دارند. البته، اگر یک مدل برای یک کار خیلی ساده باشد، ممکن است نتواند قوانین تصمیم گیری خوبی را که ظرافت های کار را در بر می گیرد، یاد بگیرد. محققان مدل ها را از طریق تکنیک های مختلف ساده تر می کنند، از جمله داشتن مدل هایی با پارامترهای کمتر یا تشویق پارامترهایی که مدل باید در اندازه کوچک با کاهش وزن باشد.

د) برای تعیین مقدار تغییر وزن در هر مرحله از آموزش شبکه MLP میتوان از رابطه روبرو استفاده نمود. به نظر شما مزایا و معایب این روش چیست؟

$$\Delta \omega = -H^{-1} \frac{\partial E}{\partial \omega}$$

$$H = \frac{\partial^2 E}{\partial \omega^2}$$

(پاسخ د)

روشی که شما اشاره کردید، یک نوع از روش‌های بهینه سازی مبتنی بر گرادیان می‌باشد که به آن روش نیوتون (Newton's method) یا روش نیوتون-رافسون (Newton-Raphson method) می‌گویند. این روش از ماتریس هسین (Hessian matrix) که دومین مشتق تابع هدف (مثلاً تابع خطا) را نسبت به پارامترها (مثلاً وزن‌ها) نشان می‌دهد، استفاده می‌کند. این روش مزایا و معایب زیر را دارد:

مزایا:

- این روش می‌تواند به سرعت به حداقل محلی (local minimum) یا حداقل جهانی (global minimum) تابع هدف برسد، زیرا می‌تواند از اطلاعات دومین مشتق برای تنظیم مقدار و جهت گام بهینه سازی استفاده کند.
- این روش می‌تواند در مواردی که تابع هدف خمیده (convex) یا نزدیک به خمیده باشد، عملکرد خوبی داشته باشد، زیرا در این صورت حداقل جهانی وجود دارد و ماتریس هسین مثبت معین (positive definite) است.

معایب:

- این روش نیاز به محاسبه ماتریس هسین و وارون آن دارد که هزینه محاسباتی زیادی دارد، به خصوص اگر تعداد پارامترها زیاد باشد.
- این روش ممکن است در مواردی که تابع هدف غیرخمیده (non-convex) باشد، عملکرد خوبی نداشته باشد، زیرا در این صورت ممکن است به نقطه چالش برانگیر (saddle point) یا حداقل محلی (local maximum) منجر شود که ماتریس هسین منفی معین (negative definite) یا نامعین (indefinite) باشد.

روش نیوتون رافسون یا روش نیوتون روشی قدرتمند برای حل عددی معادلات است. معمولاً برای تقریب ریشه‌های توابع با ارزش واقعی استفاده می‌شود. روش نیوتون رپسون توسط آیزاک نیوتون و جوزف رافسون توسعه داده شد، از این رو روش نیوتون راپسون نامگذاری شد. روش نیوتون رافسون شامل اصلاح مکرر حدس اولیه برای همگرایی آن به سمت ریشه مورد نظر است. با این حال، این روش برای محاسبه ریشه‌های چند جمله‌ای‌ها یا معادلات با درجات بالاتر کارآمد نیست، اما در مورد معادلات درجه کوچک، این روش نتایج بسیار سریعی را به همراه دارد.

یکی از سریع ترین همگرایی‌ها به ریشه به صورت درجه دوم روی ریشه همگرا می‌شود. تبدیل آسان به ابعاد مختلف می‌توان از آن برای "جلی بخشیدن" ریشه‌ای که با روش‌های دیگر یافت می‌شود استفاده کرد. این روش میتواند به سرعت به بهینه‌سازی محلی یا جهانی همگرا شود، زیرا نرخ یادگیری را بر اساس شیب تابع خطا تغییر میدهد.

- این روش میتواند از گیر کردن در نقاط ثابت یا saddle point جلوگیری کند، زیرا نرخ یادگیری را در صورت کاهش شبکه تابع خطا افزایش میدهد.
- این روش میتواند به کاهش اثر نویز یا تغییرات تصادفی در دادهها کمک کند، زیرا نرخ یادگیری را در صورت افزایش شبکه تابع خطا کاهش میدهد.
- معایب:
- این روش ممکن است به حداقل محلی نزدیک شود و از حداقل جهانی دور شود، زیرا نرخ یادگیری را در صورت کاهش شبکه تابع خطا افزایش میدهد.
- این روش ممکن است به حالت نوسان یا oscillation برود و از همگرایی دور شود، زیرا نرخ یادگیری را در صورت افزایش شبکه تابع خطا کاهش میدهد.
- این روش نیاز به محاسبه مشتق دوم تابع خطا نسبت به وزن دارد که ممکن است پیچیده و زمانبر باشد

Disadvantages of Newton-Raphson

- Must find the derivative
- Poor global convergence properties
- Dependent on initial guess
 - May be too far from local root
 - May encounter a zero derivative
 - May loop indefinitely

Advantages of Newton-Raphson

- One of the fastest convergences to the root
- Converges on the root quadratically
 - Near a root, the number of significant digits approximately doubles with each step.
 - This leads to the ability of the Newton-Raphson Method to “polish” a root from another convergence technique
- Easy to convert to multiple dimensions
- Can be used to “polish” a root found by other methods

The Newton Raphson equation is...

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)}$$

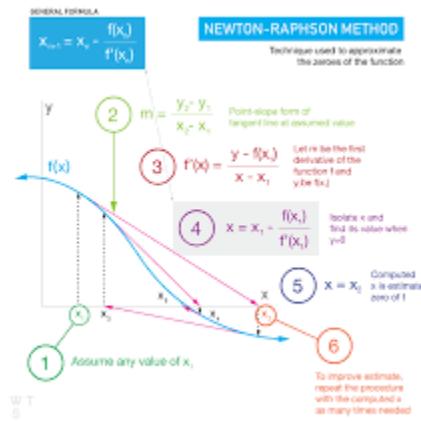
In this case I need to consider the function $y = f(x) = x - \cos(x)$
So $f'(x) = 1 + \sin x$

The Newton-Raphson formula becomes:

$$x_{n+1} = x_n - \frac{(x - \cos x)}{1 + \sin x}$$

$$x_{n+1} = \frac{x_n \sin(x_n) + \cos(x_n)}{1 + \sin x}$$

I will use a first approximation of $x_1 = 0.73$



چرا رویکرد مبتنی بر Hessian از لحاظ نظری بهتر از SGD است؟ اکنون، بهینه‌سازی مرتبه دوم با استفاده از روش نیوتن برای یافتن تکراری « X » بهینه، یک هک هوشمندانه برای بهینه‌سازی سطح خطا است، زیرا برخلاف SGD که در آن صفحه را در نقطه x_0 قرار می‌دهید و سپس پرس گام به گام را تعیین می‌کنید. در بهینه‌سازی مرتبه دوم، منحنی درجه دوم را در x_0 پیدا می‌کنیم و مستقیماً حداقل انحنا را پیدا می‌کنیم. این فوق العاده کارآمد و سریع است. ولی !!! با این حال، از نظر تجربی، آیا اکنون می‌توانید تصور کنید که یک Hessian برای شبکه‌ای با میلیون‌ها پارامتر محاسبه کنید؟ البته بسیار ناکارآمد می‌شود زیرا مقدار ذخیره سازی و محاسبات مورد نیاز برای محاسبه Hessian نیز مرتبه درجه دوم است. بنابراین، اگرچه از نظر تئوری، این عالی است، اما در عمل بسیار بد است. ما به یک هک برای هک نیاز داریم! و به نظر می‌رسد که پاسخ در Gradient های مزدوج نهفته است. ماتریس Hessian ماتریس مربعی از مشتق‌ات جزئی مرتبه دوم یک اسکالر است که انحنای محلی یکتابع چند متغیره را توصیف می‌کند. به طور خاص در مورد یک شبکه عصبی، Hessian یک ماتریس مربع است که تعداد سطرها و ستون‌ها برابر با تعداد کل پارامترهای شبکه عصبی است.

ویکرد مبتنی بر Hessian در بهینه‌سازی شبکه عصبی یک روشی است که از ماتریس دومین مشتق تابع هزینه یا ماتریس Hessian استفاده می‌کند. این روش میتواند سرعت و دقیقیت همگرایی شبکه عصبی را بهبود بخشد و از گیر کردن در کمینه‌های محلی جلوگیری کند. اما این روش هم دارای معایبی است که عبارتند از:

- محاسبه ماتریس هسین برای شبکه‌های عصبی با ابعاد بالا ممکن است بسیار پیچیده و زمانبر باشد.
- ماتریس هسین ممکن است نا مثبت تعریف شده یا نا معین باشد که باعث میشود روش بهینه‌سازی ناپایدار شود.
- ماتریس هسین ممکن است تغییرات بزرگی در مقادیر خود داشته باشد که باعث میشود روش بهینه‌سازی حساس به انتخاب پارامترهای

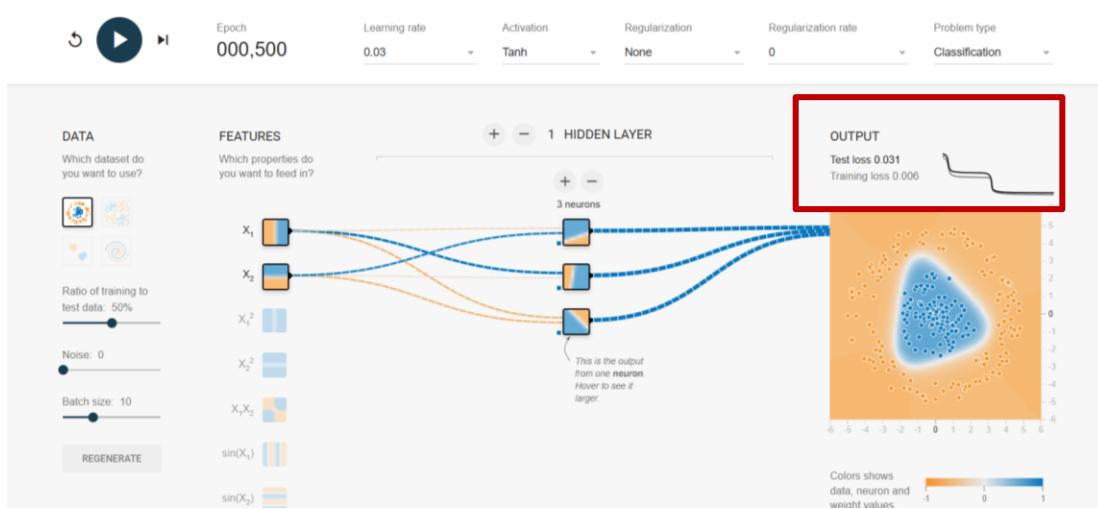
این روش میتواند جهت بهینه برای به روز رسانی وزنها و بایاسها را تعیین کند که منجر به سرعت و دقیقیت بالاتر همگرایی شبکه عصبی میشود.

- این روش میتواند از گیر کردن در کمینه‌های محلی جلوگیری کند و به دنبال کمینه‌های سراسری باشد که معمولاً مقدار تابع هزینه را کمتر میکنند.
- این روش میتواند نرخ یادگیری را برای هر وزن به طور جداگانه تنظیم کند و از انتخاب یک نرخ یادگیری ثابت یا کاهشی برای تمام وزنها بپرهیزد که ممکن است باعث کند شدن یادگیری یا پرس از کمینه‌ها شود

۳- به این لینک مراجعه کنید و مدل MLP با ۱ لایه پنهان و ۳ نورون میانی را روی هر چهار دیتا موجود در وبسایت ایجاد کنید. سپس با توابع فعالساز مختلف چندبار مدل را نهایتاً تا ۵۰۰ مرحله آموزش دهید و تاثیر این توابع فعالساز را روی هر دیتا تحلیل کنید.

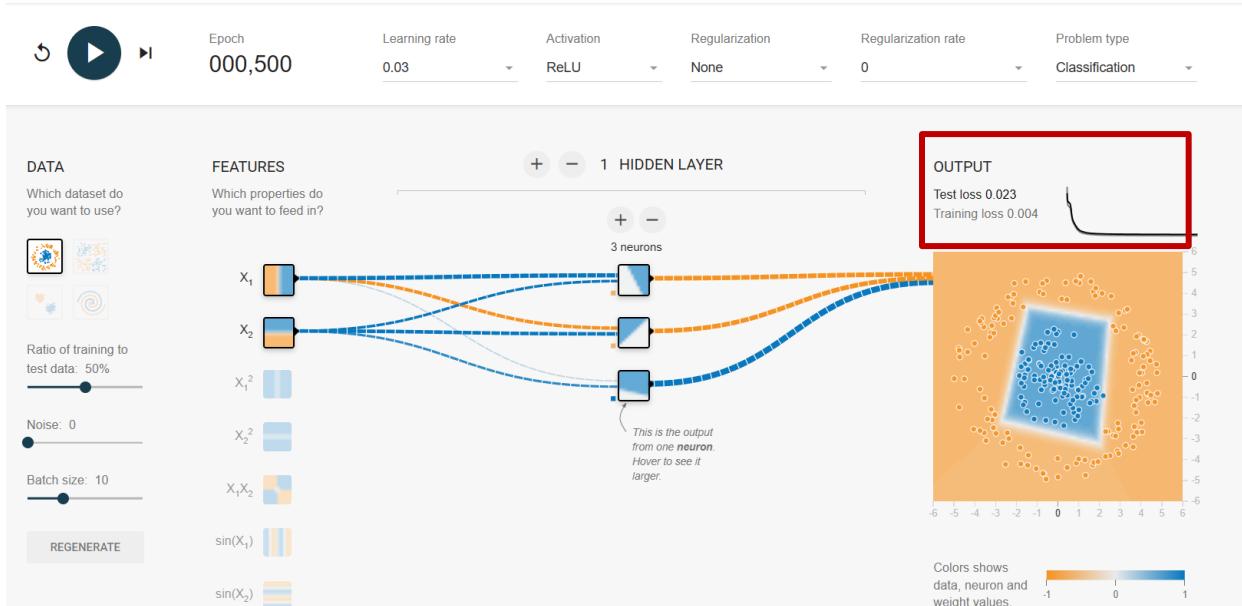
(۳) پاسخ

۱) دیتاست circle



مقدار لاس = ۰,۰۳۱

مقدار خوبی است و خوب تفکیک شده است و مرز تقریباً مثلثی تشکیل داده و محدود بین بازه ۱ و -۱ است باعث سرعت همگرایی می‌شود

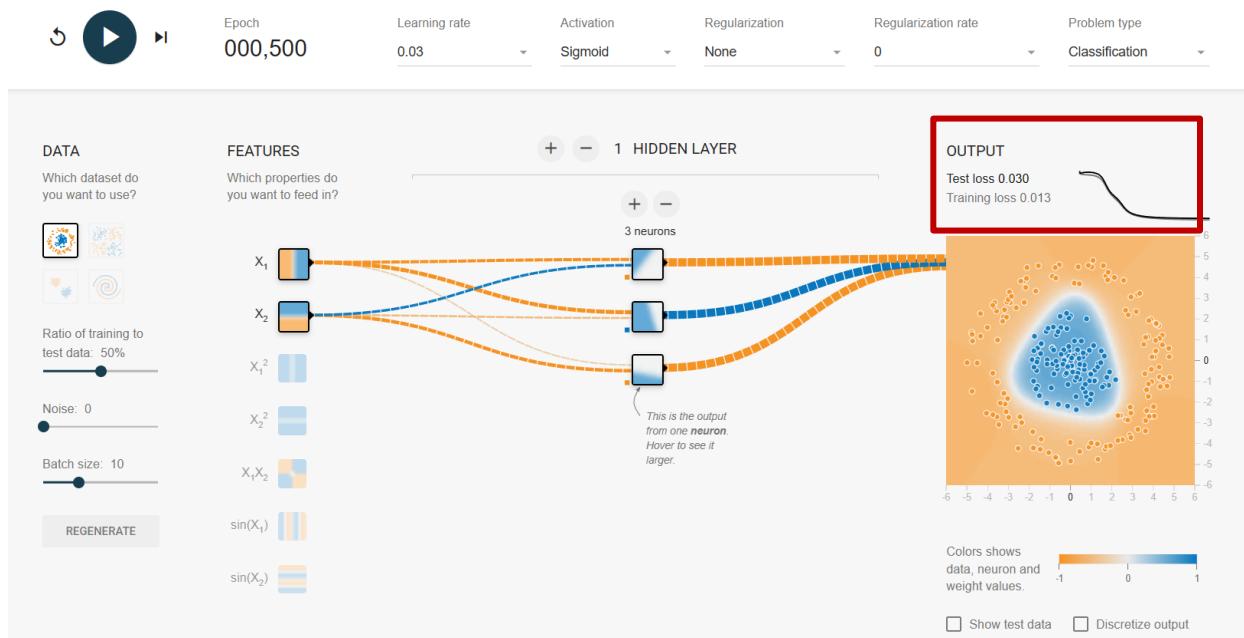


کمترین مقدار لاس بهترین تابع فعال سازی برای مجموعه داده دایره ای به نوع مدل شبکه عصبی و لایه خروجی بستگی دارد. برای لایه‌های پنهان، یک انتخاب رایج واحد خطی اصلاح شده یا ReLU است که رایج‌ترین تابع فعال‌سازی امروزه است.

یک تابع ساده است که در صورت مثبت بودن ورودی و در غیر این صورت صفر را برمی گرداند. ReLU مزیت غلبه بر مشکل گرادیان ناپدید شدن را دارد و به مدل‌ها امکان می‌دهد سریع‌تر یاد بگیرند و عملکرد بهتری داشته باشند.

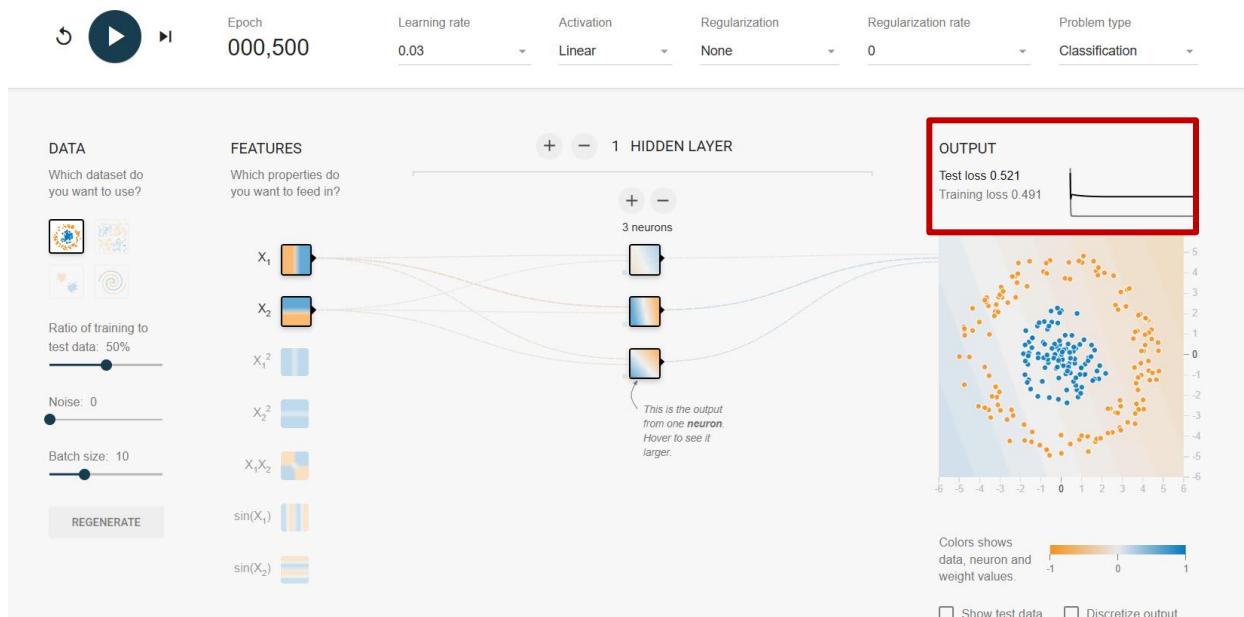
مقدار لاس = ۰,۰۲۳

به صورت ۴ ضلعی مرز مشخص شده است. همگرایی و شبیه مناسب تری دارد



مقدار لاس = ۰,۰۳۰

مقدار خوبی است و خوب تفکیک شده است و مرز تقریباً مثلثی تشکیل داده



مقدار لاس از همه موارد دیگر بیش تر است و تفکیک پذیری صورت نگرفته چون این تابع فعالسازی خطی است و نمی تواند ویژگی های غیر خطی را پوشش دهد و تفکیک پذیر و تعیین پذیری درستی داشته باشد همانگونه که در سوال ۲ اشاره کردیم به تفاوت توابع فعال سازی خطی و غیرخطی.

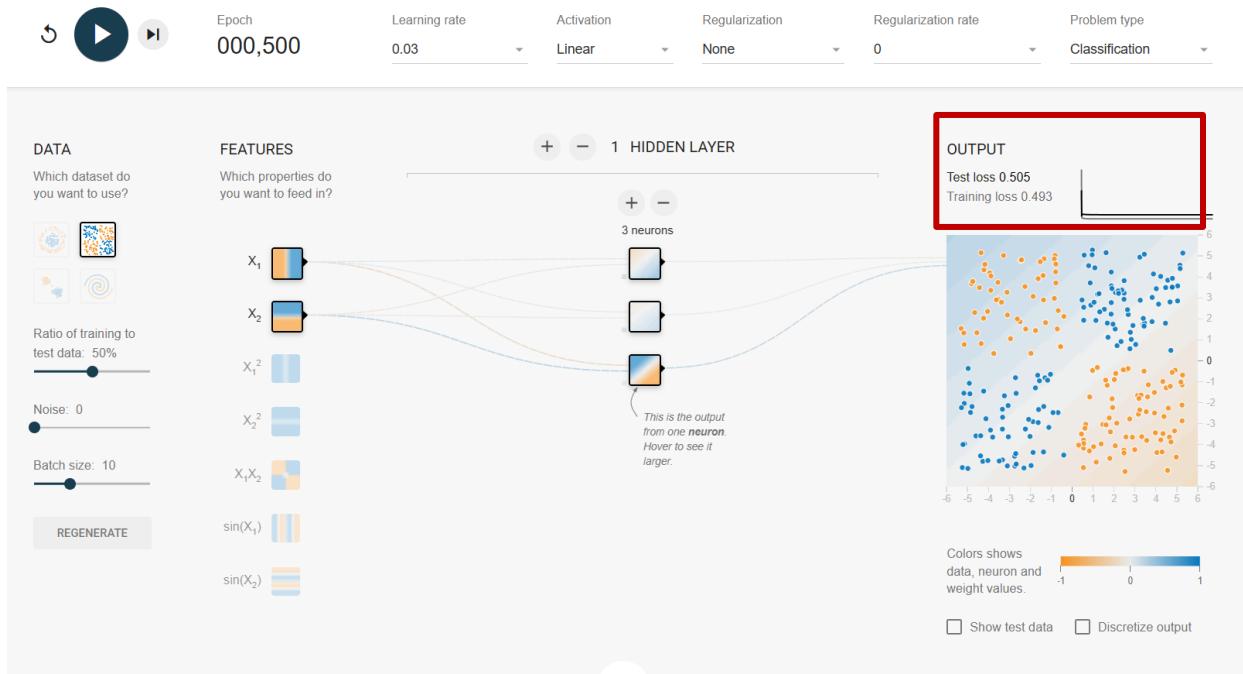
اینجا ۵۰ درصد لاس داریم یعنی به صورت کاملاً رندم هم و بدون اعمال شبکه هم در حالت عددی با امید ریاضی رندوم رو به رو خواهیم شد.

تابع خطی نمی تواند مرز دایره ای را تشخیص دهد باید از ویژگی های درجه دو استفاده کند و غیر از این تابع خطی بقیه توابع تا حد خوبی مرز را شناسایی کردن و مرز را به صورت چند ضلعی و یا دایروی شناسایی کردن.

بهترین تابع فعال سازی برای این دیتا است به ترتیب

Relu>Tan h>sigmoid>linear

Exclusive or (۲



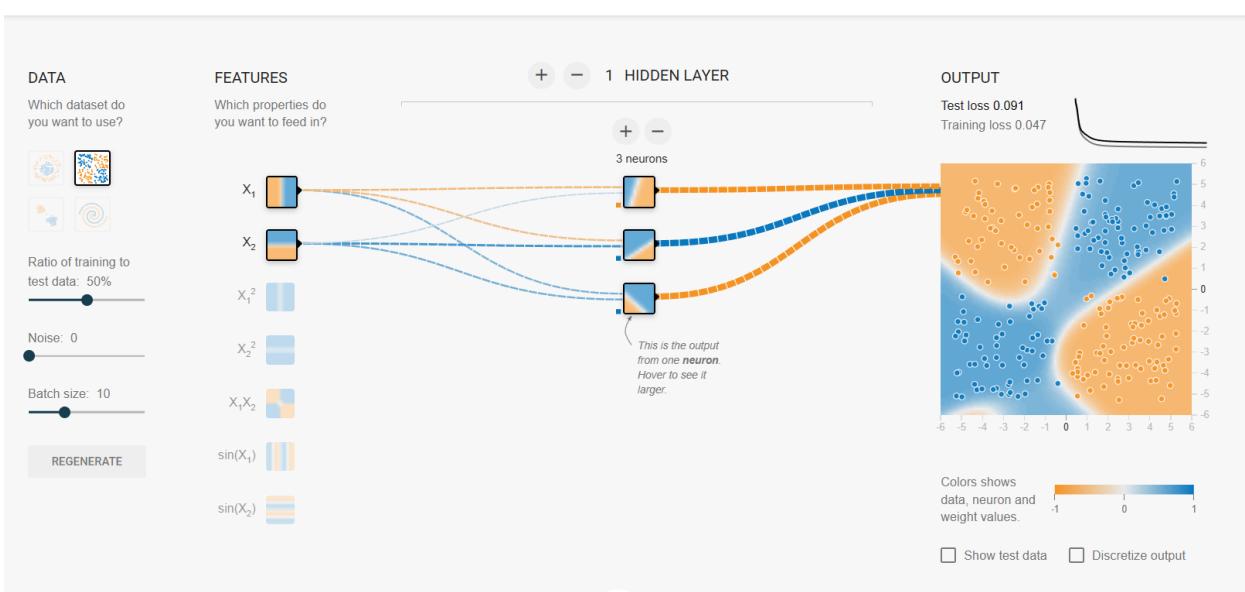
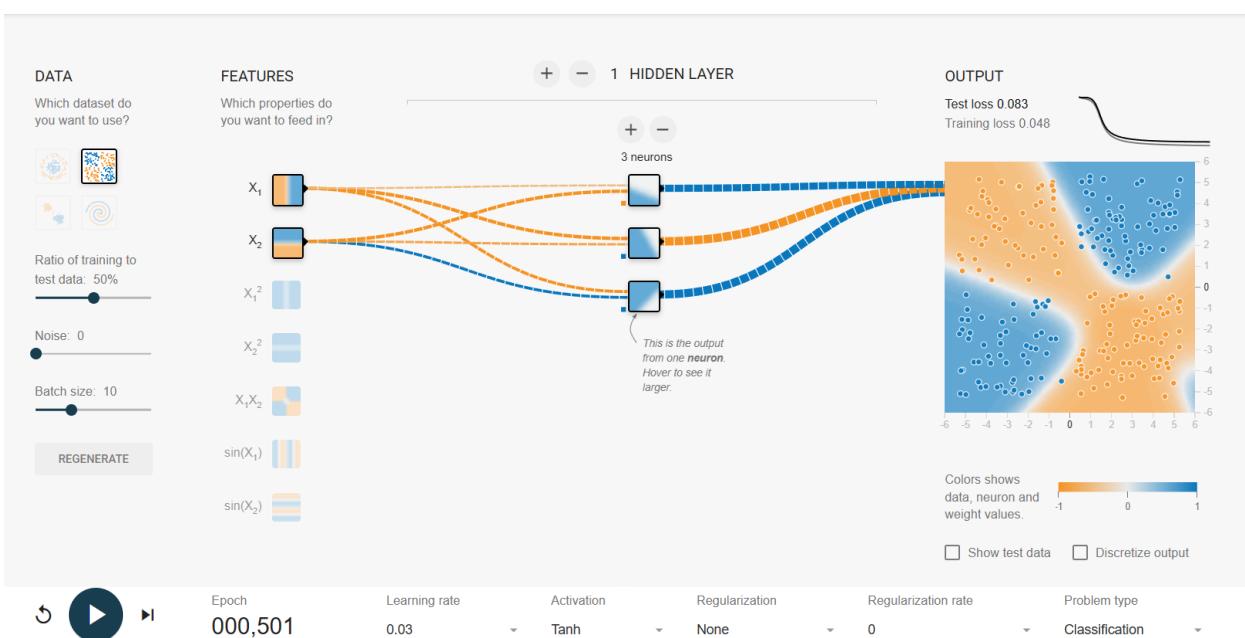
مقدار لاس = ۰,۰۵۰۵

مقدار لاس از همه موارد دیگر بیش تر است و تفکیک پذیری صورت نگرفته چون این تابع فعالسازی خطی است و نمی تواند ویژگی های غیر خطی را پوشش دهد و تفکیک پذیر و تعیین پذیری درستی داشته باشد همانگونه که در سوال ۲ اشاره کردیم به تفاوت توابع فعال سازی خطی و غیرخطی. و گویا رندوم و بدون داشتن شبکه هم به همین نتیجه برسیم

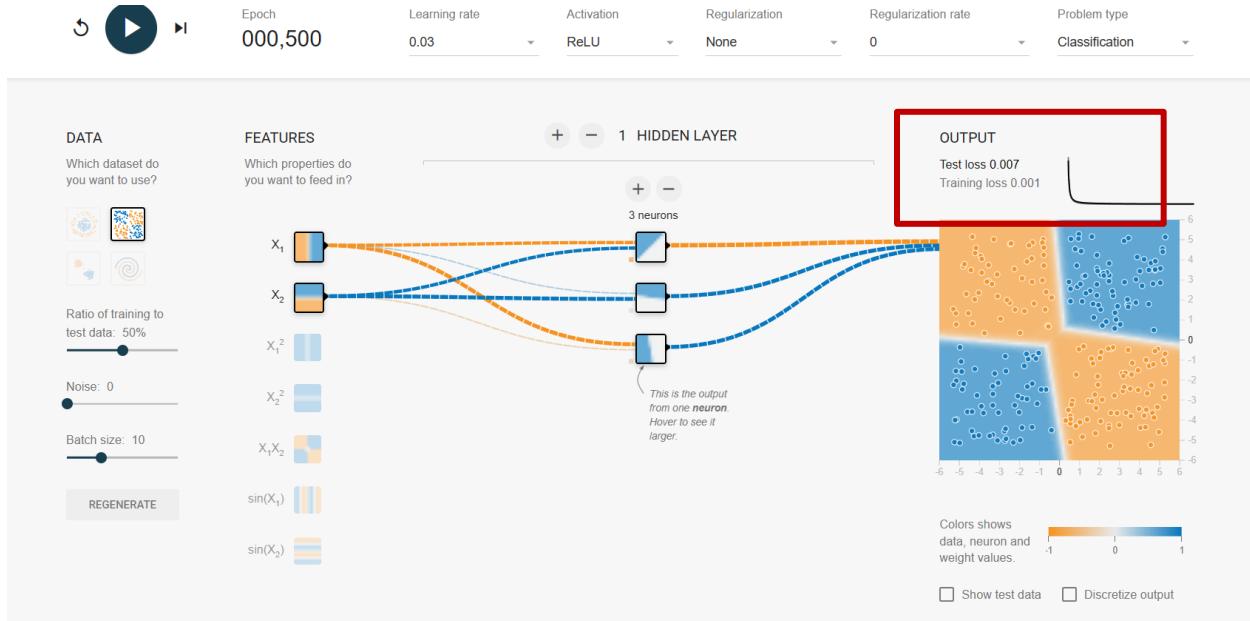
مقدار لاس = ۰,۰۵۰۵

مقدار لاس = ۰,۰۵۰۵

مقدار لاس = ۰,۰۸۳ هنوز خوب همگرا نشده در ایپوک های بعد به مرز بهتری خواهد رسید یا با نورون های بیش تر اما تا حدود خوبی توانسته شناسایی کند



مقدار لاس = ۰،۰۹۱
خوب همگرا نشده در ایپوک های بعد به مرز بهتری خواهد رسید یا با نورون های بیش تر اما تا حدود خوبی توانسته شناسایی کند



مقدار لاس = ٠,٠٠٧

نقریباً لاس به صفر رسیده و مرز ها کاملاً شناسایی شده و بهترین تابع بوده
کم ترین مقدار لاس در بین توابع فعال سازی

Relu >sigmoid> Tan h >linear

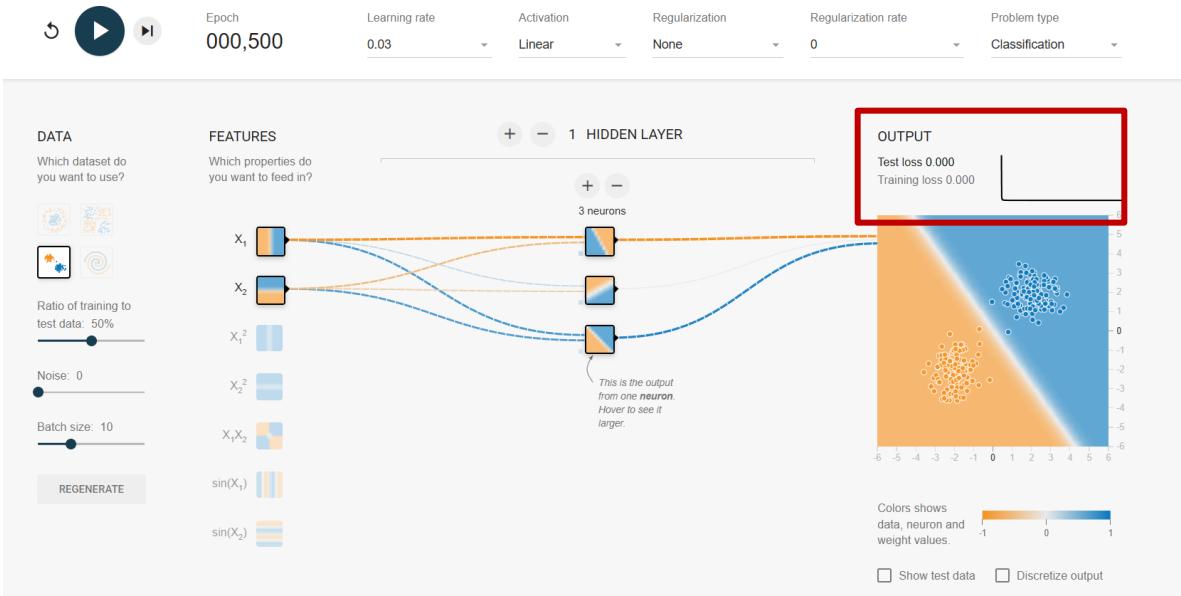
بهترین تابع فعال سازی برای یک مجموعه داده انحصاری یا (XOR) یک تابع غیرخطی است که می تواند داده ها را به دو کلاس تقسیم کند. مجموعه داده XOR نوعی داده است که دارای دو ویژگی) مختصات x و (y و دو کلاس (۰ یا ۱) است، که در آن کلاس ۱ است اگر x و y مقادیر متفاوتی داشته باشند و در غیر این صورت ۰ است. نمونه ای از مجموعه داده های XOR در زیر نشان داده شده است[!]: داده داده XOR]

یک تابع فعال سازی خطی، مانند تابع هویت یا تابع سیگموئید، نمی تواند داده ها را به دو کلاس تقسیم کند، زیرا داده ها به صورت خطی قابل تفکیک نیستند. یک تابع فعال سازی خطی یک خط مستقیم را به عنوان مرز تصمیم تولید می کند که برخی از نقاط داده را به اشتباه طبقه بندی می کند. به عنوان مثال، تابع سیگموئید یک مرز تصمیم مانند زیر تولید می کند:

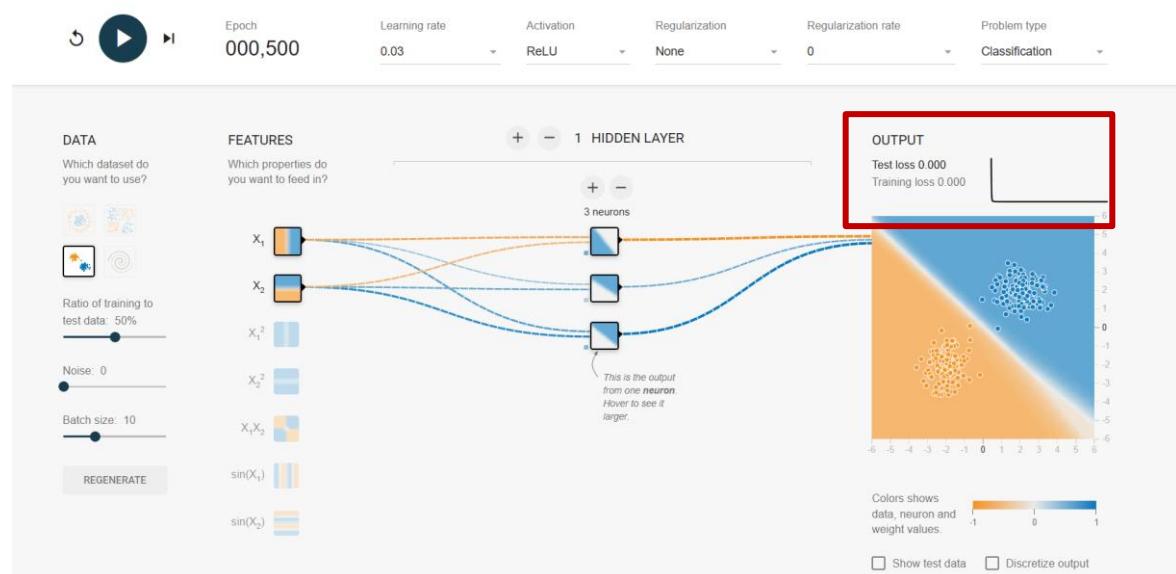
[!] عملکرد سیگموئید[

یک تابع فعال سازی غیرخطی، مانند تابع tanh یا تابع ReLU ، می تواند داده ها را به دو کلاس تقسیم کند، زیرا داده ها به صورت غیرخطی قابل تفکیک هستند. یک تابع فعال سازی غیرخطی یک خط منحنی را به عنوان مرز تصمیم تولید می کند که به درستی تمام نقاط داده را طبقه بندی می کند. به عنوان مثال، تابع tanh یک مرز تصمیم مانند زیر تولید می کند[!]: تابع Tanh بهترین تابع فعال سازی برای یک مجموعه داده XOR یک تابع غیرخطی است، مانند تابع tanh یا تابع ReLU . آین توابع می توانند الگوهای پیچیده در داده ها را بیاموزند و در وظایف طبقه بندی به خوبی عمل کنند. برای اطلاعات بیشتر در مورد عملکردهای فعال سازی و نحوه انتخاب آنها،

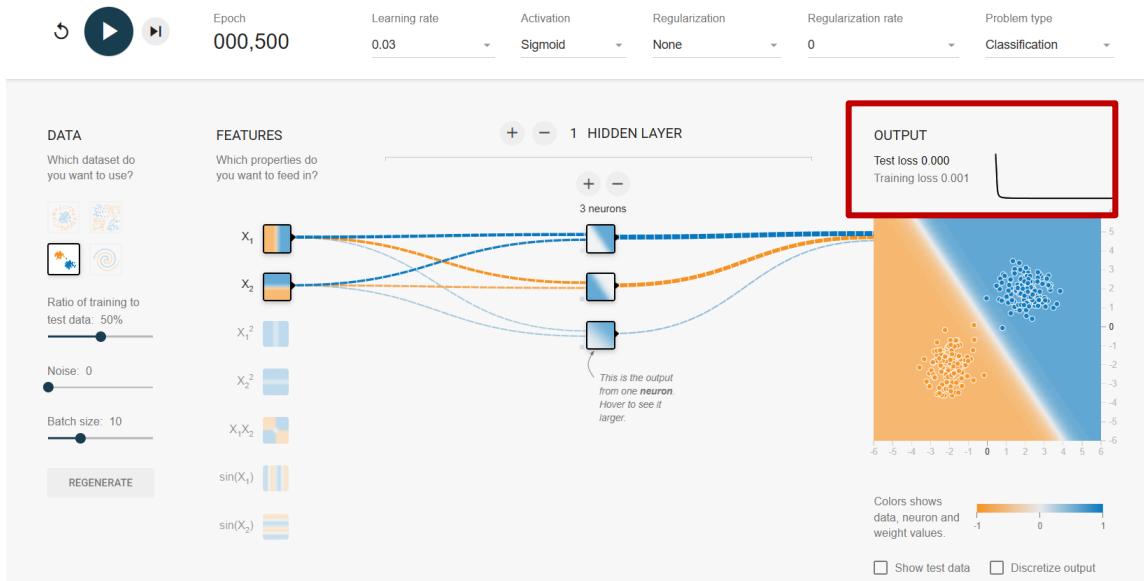
Gaussian (۳



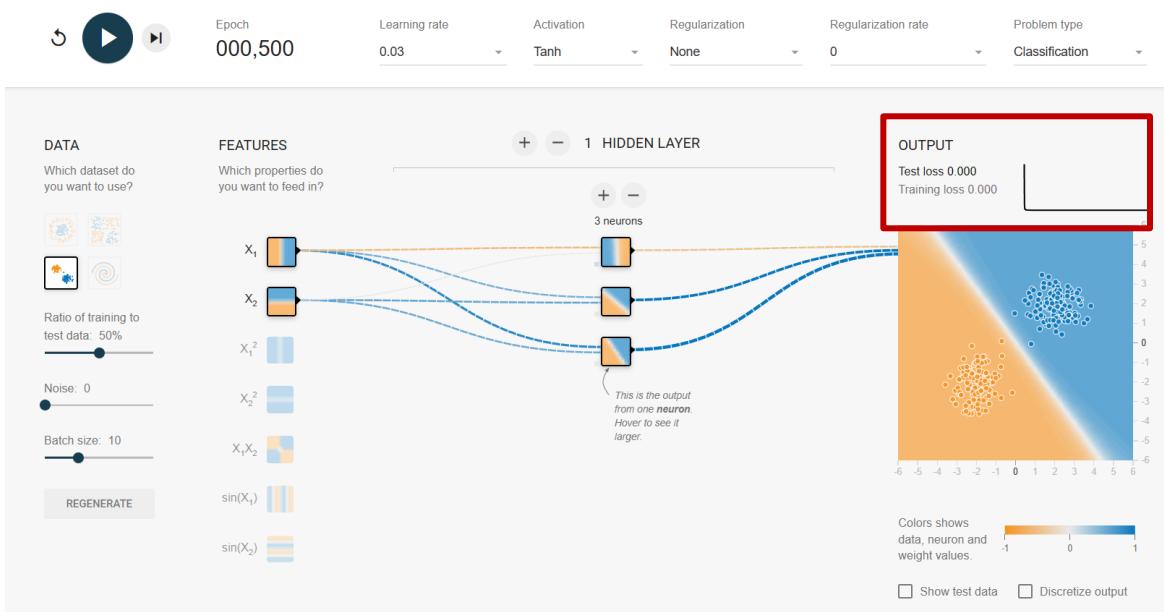
كم ترين مقدار لاس =



كم ترين مقدار لاس =



كم ترین مقدار لاس =



كم ترین مقدار لاس =

Relu=Tan h=sigmoid=linear

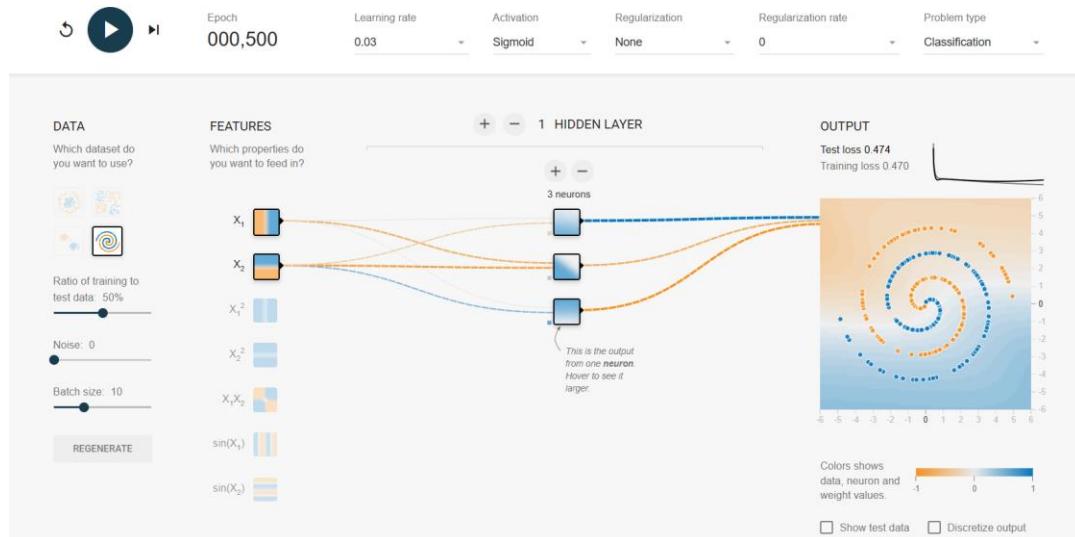
چون داده و دیتاست به دو مجموعه تقسیم می شود با یک خط ساده قابلیت جداسازی دارد و بعد از ۵۰۰ مرحله هر چهار تابع حتی تابع خطی ساده بعد از چند مرحله اموزش توانسته مرز را بسیار خوب جداسازی کند. با این که قدر برای ویژگی استخراج ویژگی های پیچیده را ندارد

داده گاوی نوعی از داده است که دارای دو ویژگی) مختصات x و y و یک یا چند کلاس است که در آن نقاط داده بر اساس توزیع نرمال یا گاوی توزیع می شوند. نمونه ای از داده های گاوی در زیر نشان داده شده است! داده های گاوی بهترین تابع فعال سازی برای داده های گاوی به نوع مدل شبکه عصبی و لایه خروجی بستگی دارد. برای لایه های پنهان، یک انتخاب رایج واحد خطی خطای گاوی یا GELU است که یک تابع فعال سازی است که به صورت $\Phi(x) = \frac{x}{1 + e^{-x}}$ تعریف می شود، که در آن $\Phi'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$ می شود. تابع GELU همچنین این مزیت را دارد که می تواند الگوهای خطی و غیرخطی را در داده ها ثبت کند، برخلاف تابع سینکوئید یا \tanh که در نهایت اشباع می شوند. نشان داده شده است که عملکرد GELU از سایر عملکردهای فعال سازی در برنامه های مختلف پردازش زبان طبیعی و دید کامپیوتری بهتر عمل می کند. برای لایه خروجی، انتخاب تابع فعال سازی به نوع مشکل پیش بینی بستگی دارد. اگر مشکل یک مشکل رگرسیونی باشد، جایی که خروجی یک مقدار پیوسته است، یک انتخاب رایج تابع هویت است که به سادگی ورودی را به عنوان خروجی برمی گرداند. تابع هویت این مزیت را دارد که ساده و خطی است و هیچ محدودیتی در محدوده خروجی اعمال نمی کند. با این حال، تابع هویت همچنین دارای مضرات حساس بودن به نقاط پرت و نویز در داده ها است. اگر مشکل یک مشکل طبقه بندی باشد، که در آن خروجی یکی از چندین کلاس ممکن است، یک انتخاب رایج تابع softmax است. تابع softmax تابعی است که یک بردار از مقادیر بین ۰ و ۱ را برمی گرداند که نشان دهنده احتمالات ورودی متعلق به هر کلاس است. تابع softmax این مزیت را دارد که قابل تفکیک و تفسیر واضح است. با این حال، تابع softmax این مضرات را نیز دارد که به نقاط پرت حساس است و در مرکز صفر قرار ندارد. بنابراین، بهترین تابع فعال سازی برای داده های گاوی به نوع مدل شبکه عصبی و لایه خروجی بستگی دارد. بسته به نوع مشکل پیش بینی، یک ترکیب ممکن برای لایه های پنهان و هویت یا softmax برای لایه خروجی است.

تفکیک پذیری این دیتاست اسان بوده و هر چهار تابع به خوبی توانسته آن لاس را به صفر برسانند.

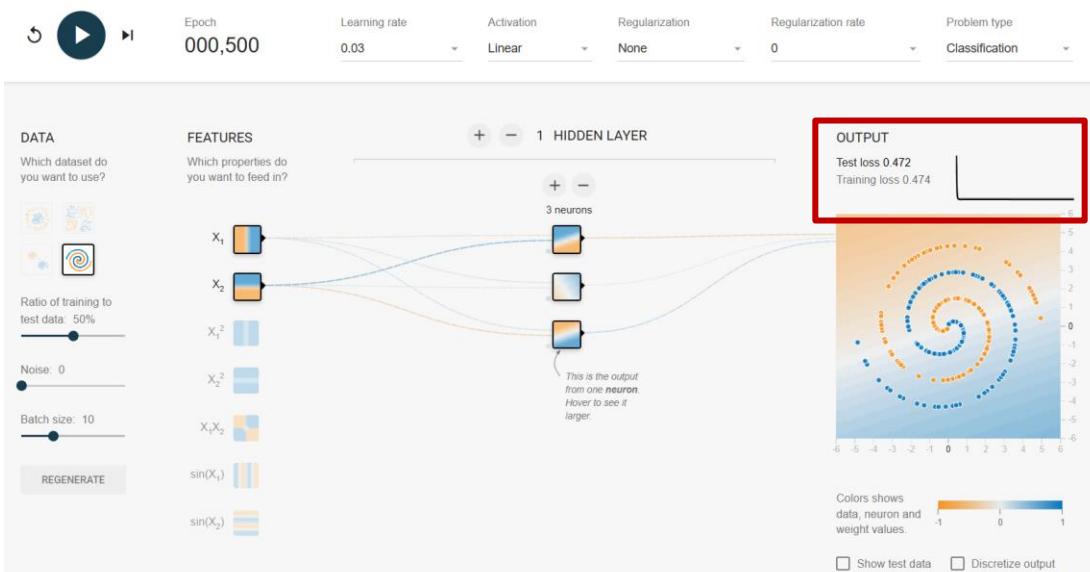
Spiral (۴)

این دیتاست بسیار پیچیده است و چند لایه ی تو در تو را شامل می شود و نمی توان با داشتن یک لایه ی نهان که تنها سه نورون دارد به نتیجه ی مطلوب رسید و مرز را پیدا کرد بلکه نیاز به لایه های پنهان بیش تر نورون های بیش تر و ویژگی های غیر خطی و استخراج انواع ویژگی ها نیاز داریم و الان پیش بینی ها به صورت رندوم هم به همین نتیجه می رسیدیم و توابع به نتیجه ای نرسیدند



مقدار لاس = ۰,۴۷۴

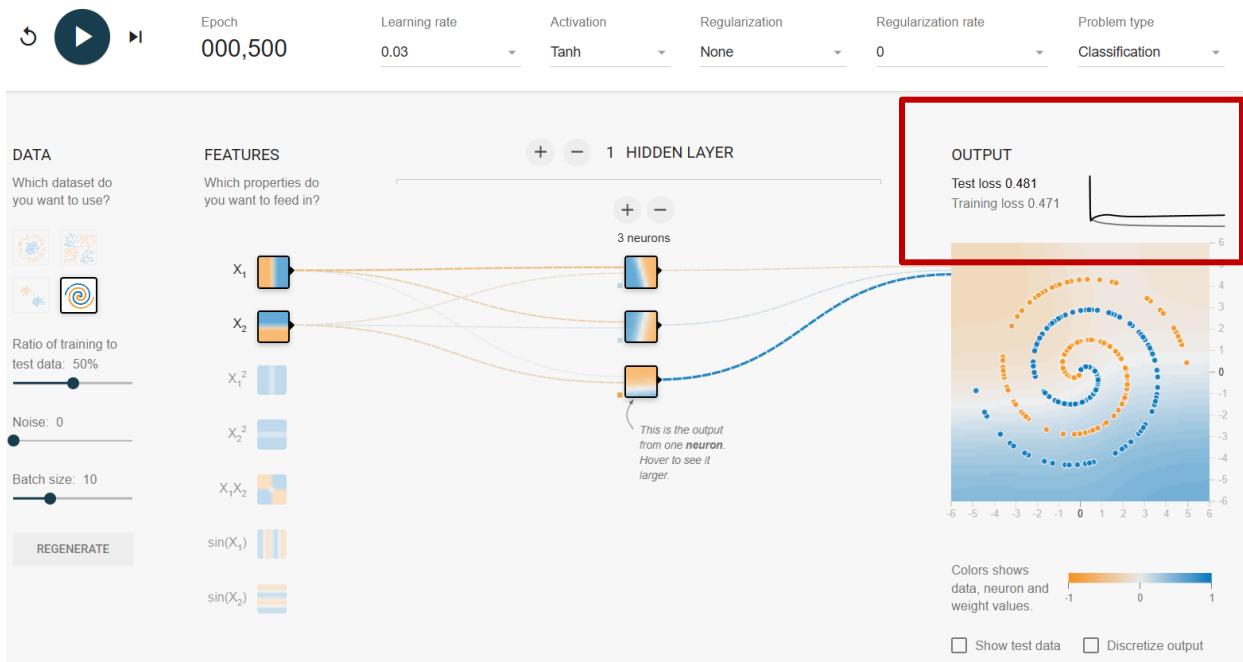
نزدیک به ۵۰ درصد لاس داریم که حالت رندوم میشود و نامناسب است



مقدار لاس = ۰,۴۷۲

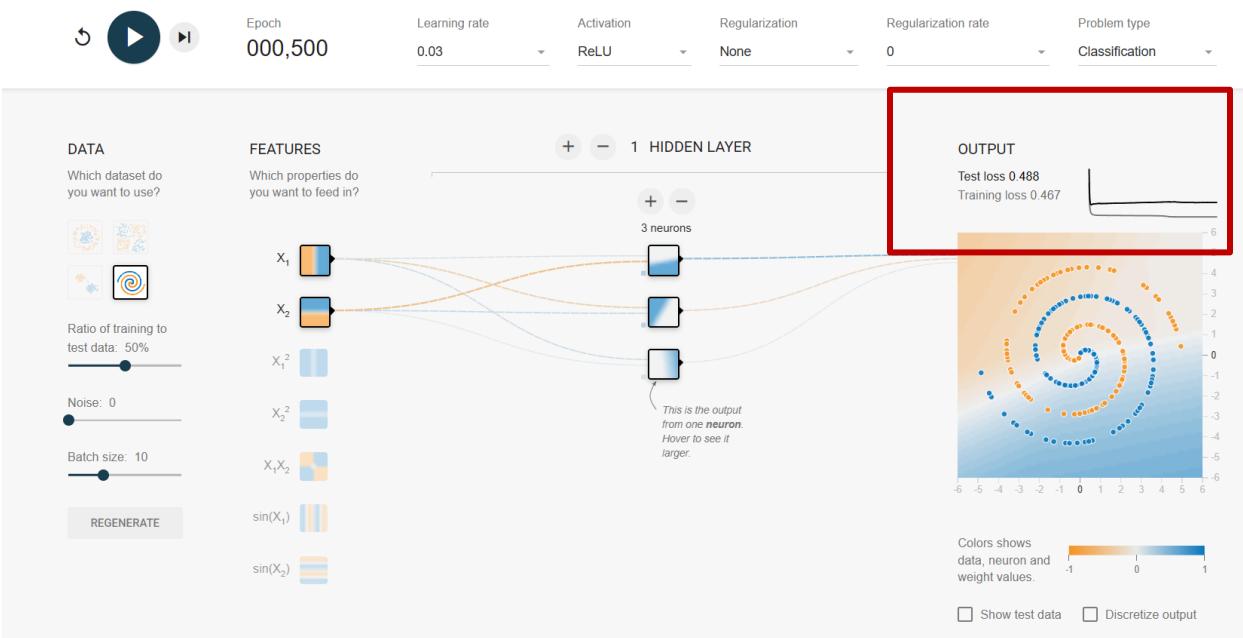
نزدیک به ۵۰ درصد لاس داریم که حالت رندوم میشود و نامناسب است

کم ترین مقدار لاس



مقدار لاس = ٠,٤٨١

نژدیک به ٥٠ درصد لاس داریم که حالت رندوم میشود و نامناسب است



مقدار لاس = ٠,٤٨٨

نژدیک به ٥٠ درصد لاس داریم که حالت رندوم میشود و نامناسب است

بدترین مقدار و بیشترین لاس

تقریبا لاس دیتا ست ها با یکدیگر برابر است و نتوانند تفکیک پذیری به خوبی صورت بگیرد حتی بعد از ۵۰۰ مرحله داده ها بسیار در یکدیگر پراکنده هستند و نتوانند جداسازی به خوبی صورت بگیرد

۴- با توجه به کامنت ها نوتبوک HW۴ را کامل کنید. از سلول ها ران بگیرید سپس کد و نمودار های رسم شده را تحلیل کنید

(پاسخ ۴)

توضیح سلول اول :

در قسمت اول من مدل را به دو روش پیاده سازی کردم
روش اول :

```
model = Sequential(  
    [  
        Dense(10, input_dim=25, name="L1"),  
        Activation("softmax"),  
    ]  
)
```

از دنس تنسورفلو برای پیاده سازی شبکه فولی کانکتد استفاده می کنیم.

- یک لایه خروجی با ۱۰ نورون و تابع فعالسازی softmax که احتمالات تعلق به هر یک از ۱۰ کلاس را محاسبه میکند. هر کلاس متناظر با یک عدد از ۰ تا ۹ است
خروجی ما ۱۰ نورونه می باشد یا ۱۰ کلاسه و تابع فعال ساز اخر را یک سافت مکس استفاده می کنم که کلاس ها را بر اساس احتمالاتشان پیش بینی کند و یک توزیع احتمالاتی داشته باشیم که کلاس با احتمال بیشتر انتخاب می شود.
تعداد نورون های ورودی یا اینچوتو ها ۲۵ تاست

حالا روشی که در آن بیشترین تابع فعال ساز را استفاده کردم توضیح می دهم:

- ین کد یک مدل شبکه عصبی چند لایه پیشرو یا MLP را با استفاده از API سلسله مراتبی یا Keras در Sequential ایجاد میکند. این مدل برای دسته بندی تصاویر دست نوشته ای اعداد ۰ تا ۹ با استفاده از مجموعه داده hoda طراحی شده است
که اعداد دست نوشته ایرانی را طبقه بندی می کند ب یکی از ۱۰ کلاس عددی ما
یک لایه ورودی شکل یا Shape را برابر با تعداد ویژگیهای داده های آموزشی تعیین میکند
۲۵ نورون و تابع فعالسازی relu که یک تابع غیرخطی است و میتواند روابط پیچیده تری را بین ورودی و خروجی بیاموزد.

- یک لایه حذف یا Dropout با نرخ ۰,۵، که به منظور جلوگیری از بیشبرازش یا overfitting به کار میرود. این لایه به طور تصادفی برخی از نورونهای لایه قبلی را حذف میکند و باعث میشود مدل به ویژگیهای مهمتر توجه کند.

- یک لایه خروجی با ۱۰ نورون و تابع فعالسازی softmax که احتمالات تعلق به هر یک از ۱۰ کلاس را محاسبه میکند. هر کلاس متناظر با یک عدد از ۰ تا ۹ است
خروجی ما ۱۰ نورونه می باشد یا ۱۰ کلاسه و تابع فعال ساز اخر را یک سافت مکس استفاده می کنم که کلاس ها را بر اساس احتمالاتشان پیش بینی کند و یک توزیع احتمالاتی داشته باشیم که کلاس با احتمال بیشتر انتخاب می شود.

```

# you code here
#####
# model = ...
# input_shape=(x_train.shape, y_train.shape,)

model = Sequential(
    [
        Input(shape=x_train.shape[1]),
        layers.Dense(25, activation="relu"),
        layers.Dropout(0.5),
        layers.Dense(10, activation="softmax"),
    ]
)

# OR
# model = Sequential(
#     [
#         Dense(10, input_dim=25, name="L1"),
#
#         Activation("softmax"),
#     ]
# )

#OR

# model = Sequential()
# model.add(Dense(10, input_dim=25, name="L1")),
# model.add(Activation("softmax")),

```

توضیح سلول دوم :

پارامتر دقت را دریافت کرده ایم متریک = دقت
 از تابع بهینه ساز آدام استفاده کردیم = یک تابع خوب با همگرایی بالا
 بهینه ساز Adam ، مخفف Adaptive Moment Estimation optimizer ، یک الگوریتم بهینه سازی است که معمولاً در یادگیری عمیق استفاده می شود . این الگوریتم توسعه‌ای از الگوریتم شبیه تصادفی (SGD) است و برای بهروزرسانی وزن‌های یک شبکه عصبی در طول تمرین طراحی شده است .

تابع ضرر برای مدل طبقه بندی چند کلاسه که در آن دو یا چند برچسب خروجی وجود دارد استفاده می شود . برچسب خروجی یک مقدار رمزگذاری یک دسته داغ به شکل ۰ و ۱ اختصاص داده می شود .
 و از لاس فانکشن categorical_crossentropy = مناسب برای طبقه بندی چند کلاسه
 • تابع compile که شبکه عصبی را برای آموزش آماده میکند . این تابع سه پارامتر مهم را مشخص میکند: تابع هزینه، الگوریتم بهینهسازی و معیار ارزیابی . در این مثال، تابع هزینه categorical_crossentropy است که میزان اختلاف بین خروجی مورد انتظار و خروجی واقعی را محاسبه میکند . الگوریتم بهینهسازی adam است که یک روش محبوب و کارآمد برای به روز رسانی وزنها و بایاسها است . معیار ارزیابی accuracy است که میزان تطابق بین برچسبهای واقعی و پیشنبینی شده را نشان میدهد .

- تابع `fit` که شبکه عصبی را بر روی داده‌های آموزشی آموزش میدهد. این تابع چندین پارامتر را دریافت میکند: داده‌های ورودی و خروجی، اندازه دسته، تعداد دوره‌ها و نسبت اعتبارسنجی. در این مثال، داده‌های ورودی و خروجی `y_train` و `x_train` هستند که شامل تصاویر و برچسبهای دست نوشته‌ی اعداد فارسی هستند. اندازه دسته ۱۲۸ است که نشان میدهد که شبکه عصبی ۱۲۸ نمونه را در هر مرحله آموزش میبیند. تعداد دوره‌ها ۱۵ است که نشان میدهد که شبکه عصبی کل داده‌های آموزشی را ۱۵ بار میبیند. نسبت اعتبارسنجی ۰,۱ است که نشان میدهد که شبکه عصبی ۱۰ درصد از داده‌های آموزشی را برای ارزیابی عملکرد خود در طول آموزش استفاده میکند

Compile model

```
# In this cell compile mode, set loss function and optimizer and get metrics accuracy
#####
# you code here
#####

batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

توضیح کلی کد :

ین کد یک شبکه عصبی چند لایه پیشرو یا MLP را با استفاده از کتابخانه کراس در پایتون ایجاد میکند. این شبکه برای دسته‌بندی تصاویر دست نوشته‌ی اعداد فارسی با استفاده از مجموعه داده Hoda طراحی شده است. این کد شامل مراحل زیر است:

- دانلود مجموعه داده Hoda از گوگل درایو با استفاده از ابزار `gdown` و فازی مچینگ
- وارد کردن کتابخانه‌های `matplotlib.pyplot`, `numpy` و `keras` برای کار با شبکه عصبی، آرایه‌ها و نمایش نمودارها
- وارد کردن مازول `dataset` که توابعی برای بارگذاری و پیشپردازش داده‌های Hoda را فراهم میکند
- تقسیم داده‌ها به دو بخش آموزشی و آزمونی و تبدیل برچسبها به بردارهای دودویی با استفاده از تابع `to_categorical`
- چاپ اطلاعات مربوط به نوع، شکل و مقدار داده‌ها قبل و بعد از پیشپردازش با استفاده از تابع `print_data_info`
- تبدیل داده‌ها به نوع `float32` و نرمال سازی آنها با تقسیم بر ۲۵۵
- تعریف معماری شبکه عصبی MLP که شامل یک لایه ورودی، یک لایه پنهان با ۲۵ نورون و تابع فعالسازی `relu`، یک لایه حذف با نرخ ۰,۵ برای جلوگیری از بیشبرازش، و یک لایه خروجی با ۱۰ نورون و تابع فعالسازی `softmax` است
- تعیین اندازه دسته، تعداد دوره‌ها، تابع هزینه، الگوریتم بهینه‌سازی و معیار ارزیابی برای شبکه عصبی

- آموزش شبکه عصبی بر روی داده‌های آموزشی با استفاده از تابع `fit` و ارزیابی شبکه عصبی بر روی داده‌های آزمونی با استفاده از تابع `evaluate`

Download dataset

```
!gdown --fuzzy https://drive.google.com/file/d/1QJrQsEY0fPBn1LoIeYMZ2HFBC0AY-6F/view?usp=sharing
!gdown --fuzzy https://drive.google.com/file/d/1zStcaVl_34RrYIfV0buM4xzB6s8xwvBi/view?usp=sharing

]
Downloading...
From: https://drive.google.com/uc?id=1QJrQsEY0fPBn1LoIeYMZ2HFBC0AY-6F
To: /content/dataset.py
100% 909/909 [00:00<00:00, 2.74MB/s]
Downloading...
From: https://drive.google.com/uc?id=1zStcaVl\_34RrYIfV0buM4xzB6s8xwvBi
To: /content/Data_hoda_full.mat
100% 3.99M/3.99M [00:00<00:00, 148MB/s]
```

Importing libraries

```
import keras
import numpy as np
import matplotlib.pyplot as plt

from keras.models import Sequential
from keras.layers import Dense, Activation
from dataset import load_hoda
from tensorflow.keras import layers
from tensorflow.keras.layers import Input, Dense
```

Cell 9 of 21 @ Go Live

Load dataset

```
x_train_original, y_train_original, x_test_original, y_test_original = load_hoda()
```

Python

Converting x_train and x_test to ndarray array format and converting y_train and y_test to one-hot-encoding:

####First, we have defined a simple function that prints the dimensions, data type and information of the loaded dataset. We will print this information before and after data preprocessing to notice the changes!

```
# Preprocess input data for Keras.  
x_train = np.array(x_train_original)  
y_train = keras.utils.to_categorical(y_train_original, num_classes=10)  
x_test = np.array(x_test_original)  
y_test = keras.utils.to_categorical(y_test_original, num_classes=10)
```

Python

```
print(x_train.shape)
```

Python

```
(1000, 25)
```

Cell 9 of 21 ⚡ Go Live ⚡ kite: not installed ✅ Prettier ⚡ { }

```
def print_data_info(x_train, y_train, x_test, y_test):  
    #Check data Type  
    print("\ttype(x_train): {}".format(type(x_train)))  
    print("\ttype(y_train): {}".format(type(y_train)))  
  
    #check data Shape  
    print("\tx_train.shape: {}".format(np.shape(x_train)))  
    print("\ty_train.shape: {}".format(np.shape(y_train)))  
    print("\tx_test.shape: {}".format(np.shape(x_test)))  
    print("\ty_test.shape: {}".format(np.shape(y_test)))  
  
    #sample data  
    print("\ty_train[0]: {}".format(y_train[0]))  
  
  
print("Before Preprocessing:")  
print_data_info(x_train_original, y_train_original, x_test_original, y_test_original)  
print("After Preprocessing:")  
print_data_info(x_train, y_train, x_test, y_test)
```

```
Before Preprocessing:  
    type(x_train): <class 'numpy.ndarray'>  
    type(y_train): <class 'numpy.ndarray'>  
    x_train.shape: (1000, 25)  
    y_train.shape: (1000,)  
    x_test.shape: (200, 25)  
    y_test.shape: (200,)  
    y_train[0]: 6  
After Preprocessing:
```

Cell 9 of 21 ⚡ Go Live

```
x_test.shape: (200, 25)
y_test.shape: (200,)
y_train[0]: 6
After Preprocessing:
type(x_train): <class 'numpy.ndarray'>
type(y_train): <class 'numpy.ndarray'>
x_train.shape: (1000, 25)
y_train.shape: (1000, 10)
x_test.shape: (200, 25)
y_test.shape: (200, 10)
y_train[0]: [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 25)	650
dropout (Dropout)	(None, 25)	0
dense_1 (Dense)	(None, 10)	260

=====

Total params: 910 (3.55 KB)
Trainable params: 910 (3.55 KB)
Non-trainable params: 0 (0.00 Byte)

=====

```
Epoch 1/15
8/8 [=====] - 1s 40ms/step - loss: 2.5364 - accuracy: 0.0833 - val_loss: 2.3788 - val_accuracy: 0.091
Epoch 2/15
8/8 [=====] - 0s 8ms/step - loss: 2.4504 - accuracy: 0.1067 - val_loss: 2.3293 - val_accuracy: 0.1201
Epoch 3/15
8/8 [=====] - 0s 7ms/step - loss: 2.3931 - accuracy: 0.1267 - val_loss: 2.2871 - val_accuracy: 0.1301
Epoch 4/15
8/8 [=====] - 0s 8ms/step - loss: 2.3483 - accuracy: 0.1233 - val_loss: 2.2495 - val_accuracy: 0.1601
Epoch 5/15
8/8 [=====] - 0s 7ms/step - loss: 2.3180 - accuracy: 0.1389 - val_loss: 2.2150 - val_accuracy: 0.2301
Epoch 6/15
8/8 [=====] - 0s 10ms/step - loss: 2.2813 - accuracy: 0.1533 - val_loss: 2.1814 - val_accuracy: 0.231
Epoch 7/15
8/8 [=====] - 0s 14ms/step - loss: 2.2313 - accuracy: 0.1544 - val_loss: 2.1509 - val_accuracy: 0.251
Epoch 8/15
8/8 [=====] - 0s 11ms/step - loss: 2.2196 - accuracy: 0.1744 - val_loss: 2.1208 - val_accuracy: 0.281
Epoch 9/15
8/8 [=====] - 0s 15ms/step - loss: 2.1849 - accuracy: 0.1878 - val_loss: 2.0901 - val_accuracy: 0.291
Epoch 10/15
8/8 [=====] - 0s 13ms/step - loss: 2.1423 - accuracy: 0.1911 - val_loss: 2.0570 - val_accuracy: 0.321
Epoch 11/15
8/8 [=====] - 0s 11ms/step - loss: 2.1191 - accuracy: 0.2233 - val_loss: 2.0244 - val_accuracy: 0.331
Epoch 12/15
8/8 [=====] - 0s 13ms/step - loss: 2.0843 - accuracy: 0.2367 - val_loss: 1.9917 - val_accuracy: 0.431
Epoch 13/15
...
Epoch 14/15
8/8 [=====] - 0s 10ms/step - loss: 2.0060 - accuracy: 0.2833 - val_loss: 1.9244 - val_accuracy: 0.481
Epoch 15/15
8/8 [=====] - 0s 10ms/step - loss: 1.9924 - accuracy: 0.2778 - val_loss: 1.8895 - val_accuracy: 0.491
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Cell 9 of 21 ⚡ Go Live ⚡ kite: not installed ⚡

Fit model

```
MLP_model = model.fit(x_train, y_train,
                       epochs=100,
                       batch_size=64, validation_data=(x_test, y_test))
```

Python

```
Epoch 1/100
16/16 [=====] - 0s 8ms/step - loss: 1.9600 - accuracy: 0.3030 - val_loss: 1.8185 - val_accuracy: 0.51
Epoch 2/100
16/16 [=====] - 0s 5ms/step - loss: 1.8615 - accuracy: 0.3460 - val_loss: 1.7417 - val_accuracy: 0.61
Epoch 3/100
16/16 [=====] - 0s 6ms/step - loss: 1.8030 - accuracy: 0.4000 - val_loss: 1.6669 - val_accuracy: 0.61
Epoch 4/100
16/16 [=====] - 0s 6ms/step - loss: 1.7331 - accuracy: 0.4060 - val_loss: 1.5936 - val_accuracy: 0.61
Epoch 5/100
16/16 [=====] - 0s 5ms/step - loss: 1.6641 - accuracy: 0.4540 - val_loss: 1.5204 - val_accuracy: 0.71
Epoch 6/100
16/16 [=====] - 0s 5ms/step - loss: 1.6215 - accuracy: 0.4760 - val_loss: 1.4502 - val_accuracy: 0.71
Epoch 7/100
16/16 [=====] - 0s 4ms/step - loss: 1.5577 - accuracy: 0.4890 - val_loss: 1.3810 - val_accuracy: 0.71
Epoch 8/100
16/16 [=====] - 0s 5ms/step - loss: 1.5387 - accuracy: 0.4980 - val_loss: 1.3182 - val_accuracy: 0.71
Epoch 9/100
16/16 [=====] - 0s 4ms/step - loss: 1.4964 - accuracy: 0.4970 - val_loss: 1.2612 - val_accuracy: 0.71
Epoch 10/100
16/16 [=====] - 0s 5ms/step - loss: 1.4482 - accuracy: 0.4990 - val_loss: 1.2094 - val_accuracy: 0.81
Epoch 11/100
16/16 [=====] - 0s 5ms/step - loss: 1.3960 - accuracy: 0.5460 - val_loss: 1.1577 - val_accuracy: 0.81
```

Cell 9 of 21 ⚡ Go Live ⚡ kite: not installed ⚡

```

Epoch 99/100
16/16 [=====] - 0s 8ms/step - loss: 0.6526 - accuracy: 0.7720 - val_loss: 0.3634 -
Epoch 100/100
16/16 [=====] - 0s 6ms/step - loss: 0.6230 - accuracy: 0.7780 - val_loss: 0.3632 -
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```

```

# plot Loss
plt.plot(MLP_model.history["loss"])
plt.plot(MLP_model.history["val_loss"])

[<matplotlib.lines.Line2D at 0x797bb43b45b0>]

```

توضیح نمودارها و تحلیل نتایج :

همانگونه که میبینیم با جلورفتن مراحل لاس کاهش یافته و دقت افزایش پیدا کرده است .
نمودار لاس نزولی بوده و نمودار دقت صعودی بوده و همانگونه که مشاهد می کنیم با جلو رفتن ایپوک ها و از ایپوک یک تا ۱۰۰ :

نمودار loss
از ۰,۶۲۳ به ۱,۹۶
نمودار val_loss
از ۰,۳۶ به ۰,۵
نمودار accuracy
از ۰,۷۷۸۰ به ۰,۷۰۳۰
نمودار val_accuracy
از ۰,۹۱۰۰ به ۰,۵۶۵۰

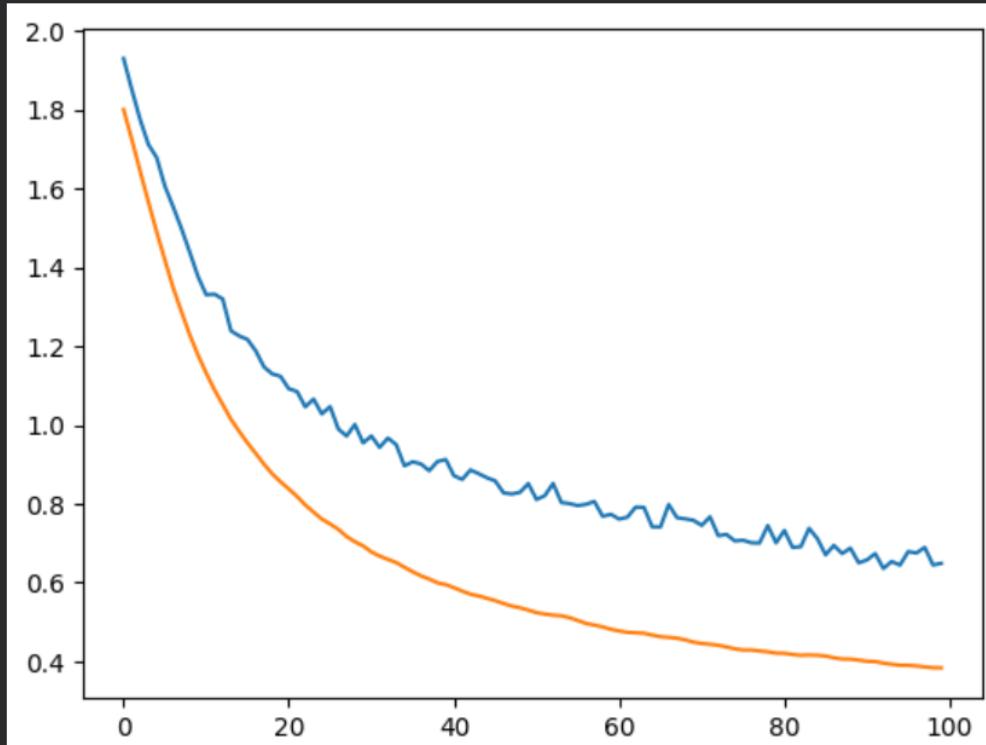
دقت داده‌ی اموزشی و تست هر دو زیاد شده و لاس داده‌ی اموزشی و تست کم شده است.

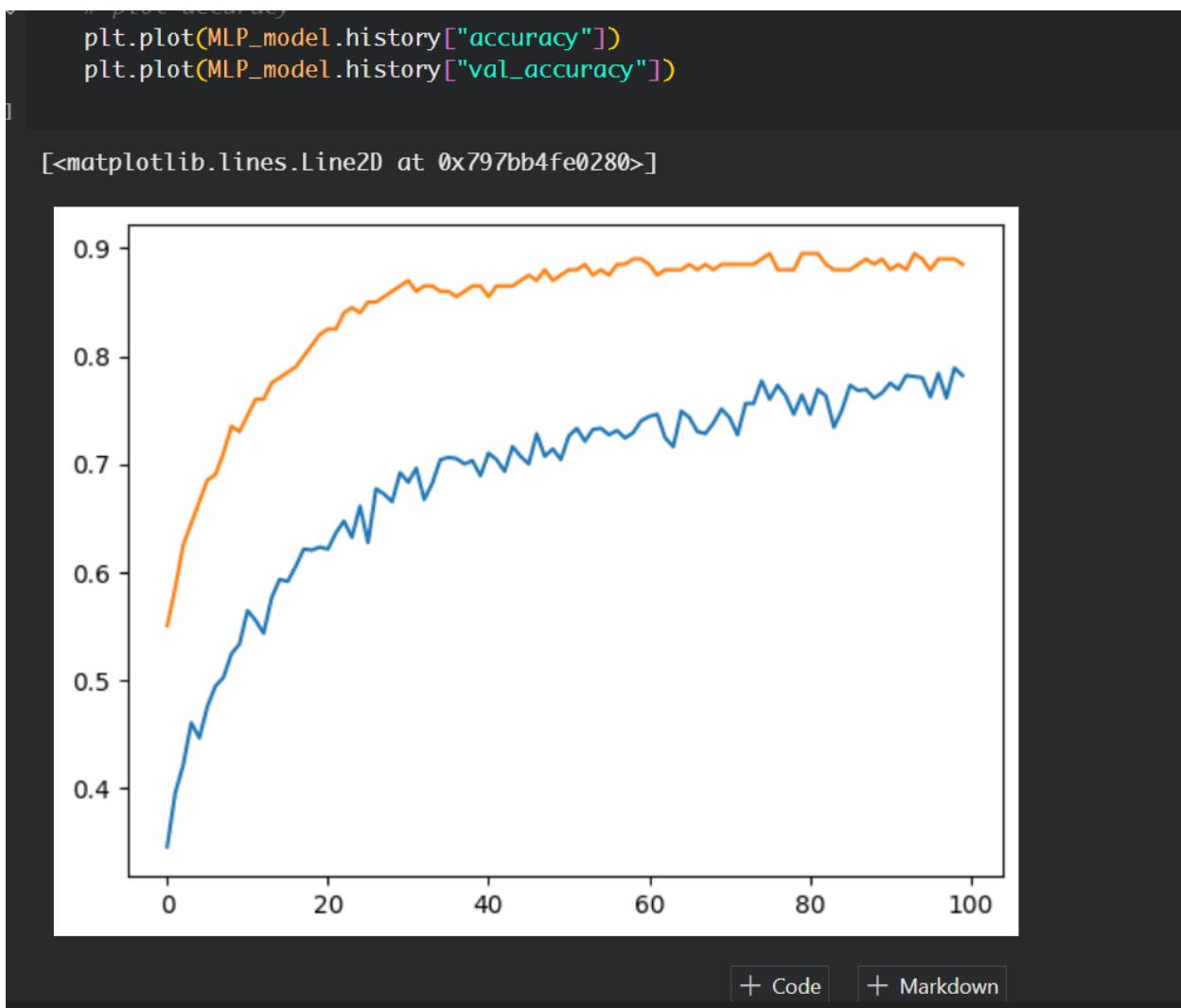
چون از قسمت تست به خوبی نتیجه داده و درصد بالایی داریم ۹۰ درصد مشخص است که under شبکه لرن داشته باشد هرچند داشتن لرنینگ ریت و تنظیم ابر پارامترها و یا استفاده از شبکه‌های عمیق تر و پیچیده تر میتواند این مسئله طبقه‌بندی را نیز حل کند و دقت ما را به یک و لاس را به صفر برساند اما اختلاف کم نمودارها نشان از عملکرد درست این شبکه نشان می‌دهد و قسمت فیت شدن شبکه به خوبی فیت شده است.

лас ماهم به حدود ۴۰ درصد رسیده و به طور اکید نزولی بوده است و اورفیت هم نشده و تا حدود خوبی توانسته که تعمیمیم بددهد و generalization صورت بگیرد(با افزایش داده‌ها در دیتابست و یا دراپ اوت این مقدار بهبود نیز میابد) چون مدل علاوه بر روی داده‌های اموزشی روی داده‌های تست هم خوب عمل کرده است. البته که نمودارها کمی باهم اختلاف دارند ولی قابل چشم پوشی است و یک فلوی یکسان را پیش گرفته‌اند.

```
# plot Loss
plt.plot(MLP_model.history["loss"])
plt.plot(MLP_model.history["val_loss"])

[]
```

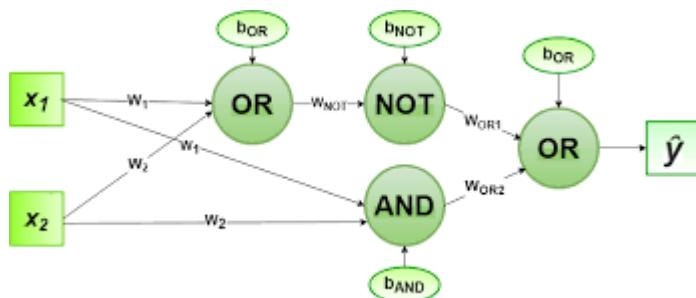


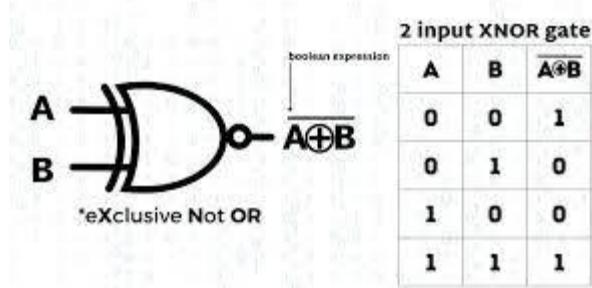


+ Code

+ Markdown

۵- با کمک کتابخانه NumPy یک پرسپترون چند لایه آموزش دهید که تابع XNOR را یاد بگیرد.
پاسخ (۵)

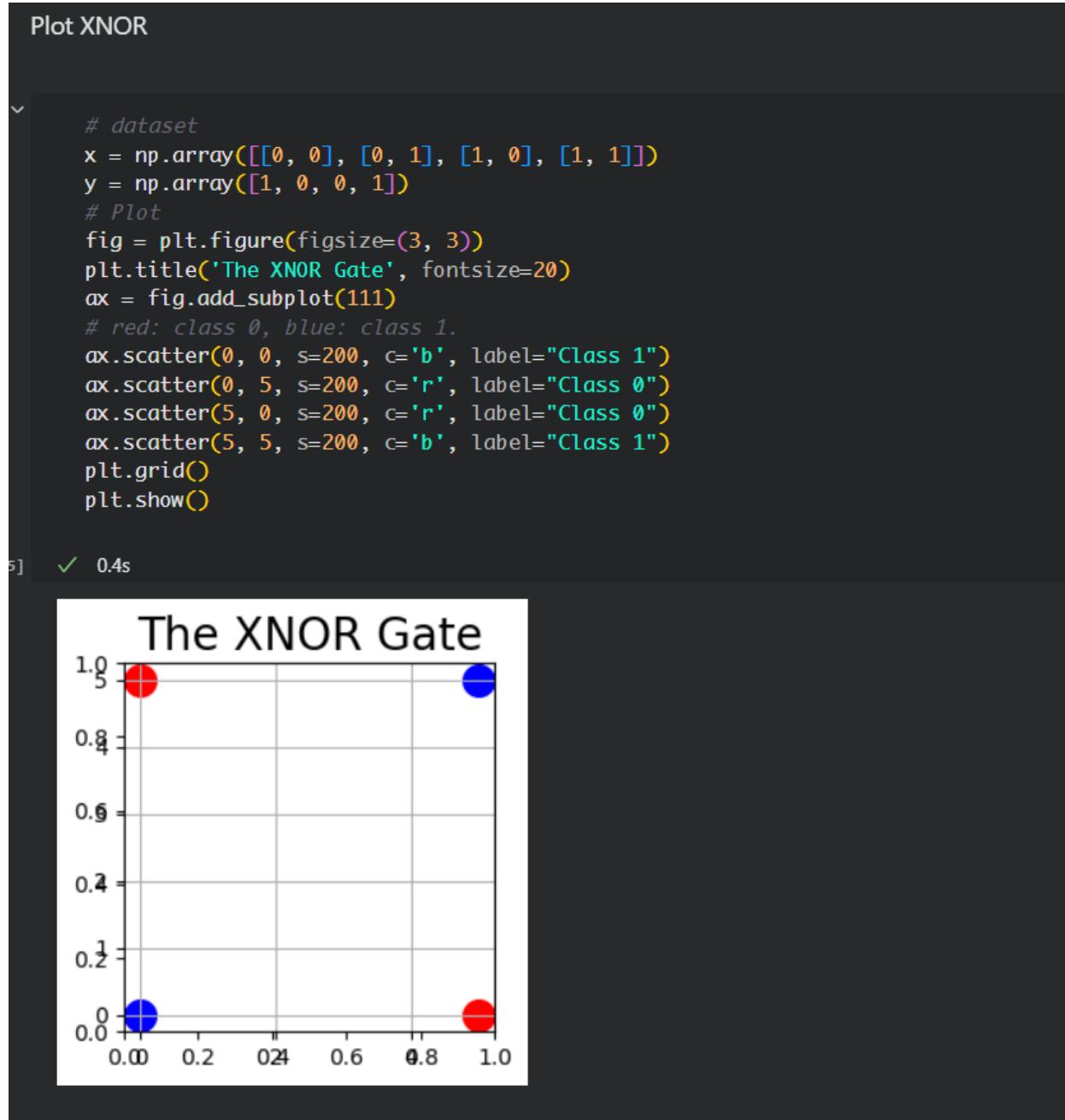




در ابتدا توابعی پایه ای که برای حل این سوال نیاز داریم را تعریف می کنیم. تابع فعال سازی استفاده شده در این مثال، sigmoid بوده که یک تابع با همان نام در کدامان تعریف می کنیم. همچین تابع محاسبه ضرمان، error entropy cross باشد که در کدامان با نام CEE تعریف شده است. سپس چون برای فاز propagation-back به مشتق تابع اور و سیگموید نیاز داریم، تابعی برای محاسبه این دو نیز می نویسیم که با نام dCEE و dsigmoid در کدامان وجود دارند. توابعی^r که تا اینجا توضیح داده شد، در قطعه کد زیر آورده شده اند.

سپس در این بخش، یک دیتاست تعریف کرده و آن را plot می کنیم. می دانیم که گیت XNOR در صورت غیر شبیه بودن بیت همه ورودی ها، ۰ و در غیر این صورت ۱ می شود. کد زده شده در این بخش به همراه نتیجه حاصله در شکل زیر آورده شده است.

اول در اینجا یک دیتاست مطابق با لاجیک XNor درست کردم و ۰۰ و ۱۱ را لیبل ۱ می‌گیرند و بقیه صفر حالا پلات ان را میبینم در مختصات ها .



توضیحات لازم در کد اورده شده است اما ابتدا کتابخانه های مورد نیاز را ایمچورت می کنیم و دو وزن رندهم را اساین می کنیم سپس مقدار سیگموید و مشتق آن را محاسبه می کنیم که برای اصلاح وزن ها و بک پروپگیشن استفاده می شود.

Creating function for running and testing NN after training

Feed forward through three layers in NN

Results are returned in normalized form in appropriate dimensions

Feeding our NN with training set and calculating output
 Feed forward through three layers in NN
 With 'numpy' library and function 'dot' we multiply matrices
 with values in layers to appropriate weights
 Results are returned in normalized form in appropriate
 dimensions
 Creating function for training the NN
 Using Backpropagation for calculating values to correct weights
 Calculating an error for output layer (Layer 2) which is the
 difference between desired output and obtained output
 We subtract matrix 4x1 of received outputs from matrix 4x1 of
 desired outputs

Showing the error each 500 iterations to track the improvements

و بعد دلتا را برای لایه‌ی خروجی و اررو را محاسبه می‌کنیم و وزن‌ها را اصلاح می‌کنیم
 سپس مدل با مجموعه‌ی اموزشی ما که لیبل‌های مشخص دارد تا ۵۰۰۰ مرحله اموزش می‌بیند و بعد روی داده‌ی تست آن را
 تست می‌کنیم .

نتیجه :

```

Creating a three layers NN by using mathematical 'numpy' library Using methods from the library to operate with matrices Importing 'matplotlib' library to plot experimental results in form of figures Importing
'numpy' library

> <
import numpy as np
import matplotlib.pyplot as plt

class NeuralNetwork():
    def __init__(self):
        np.random.seed(1)

        self.weights_0_1 = 2 * np.random.random((3, 4)) - 1
        self.weights_1_2 = 2 * np.random.random((4, 1)) - 1

        self.layer_2 = np.array([])

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def derivative_of_sigmoid(self, x):
        return x * (1 - x)

    def run_nn(self, set_of_inputs):
        layer_0 = set_of_inputs
        layer_1 = self.sigmoid(np.dot(layer_0, self.weights_0_1))
        layer_2 = self.sigmoid(np.dot(layer_1, self.weights_1_2))
        return layer_2
  
```

```

def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def derivative_of_sigmoid(self, x):
    return x * (1 - x)

def run_nn(self, set_of_inputs):
    layer_0 = set_of_inputs
    layer_1 = self.sigmoid(np.dot(layer_0, self.weights_0_1))
    layer_2 = self.sigmoid(np.dot(layer_1, self.weights_1_2))
    return layer_2

def training_process(self, set_of_inputs_for_training, set_of_outputs_for_training, epochs):
    errors = []
    for i in range(epochs):

        layer_0 = set_of_inputs_for_training
        layer_1 = self.sigmoid(np.dot(layer_0, self.weights_0_1))
        self.layer_2 = self.sigmoid(np.dot(layer_1, self.weights_1_2))

        layer_2_error = set_of_outputs_for_training - self.layer_2
        errors.append(np.mean(np.abs(layer_2_error)))

        if (i % 500) == 0:
            print('Final error after', i, 'epoch =', np.mean(np.abs(layer_2_error)))

    # Calculating delta for output layer (Layer 2)
    # Using sign '*' instead of function 'dot' of numpy library

    delta_2 = layer_2_error * self.derivative_of_sigmoid(self.layer_2)

```

Ln 18, Col 1 Spaces: 4 CRLF Cell 6 of

```

if (i % 500) == 0:
    print('Final error after', i, 'epoch =', np.mean(np.abs(layer_2_error)))

# Calculating delta for output layer (Layer 2)
# Using sign '*' instead of function 'dot' of numpy library

delta_2 = layer_2_error * self.derivative_of_sigmoid(self.layer_2)

# Calculating an error for hidden layer (Layer 1)
# Multiplying delta_2 by weights between hidden layer and output layer

layer_1_error = np.dot(delta_2, self.weights_1_2.T)

# Calculating delta for hidden layer (Layer 1)
delta_1 = layer_1_error * self.derivative_of_sigmoid(layer_1)

# Implementing corrections of weights
self.weights_1_2 -= np.dot(layer_1.T, delta_2)
self.weights_0_1 -= np.dot(layer_0.T, delta_1)

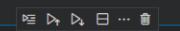
return errors

```

✓ 0.0s

Python

Create matrix 1x4 and transpose it to matrix 4x1 at the same time Creating three layers NN by initializing of instance of the class Starting the training process with data above and number of repetitions of 5000 Showing the output results after training After training process was finished and weights are adjusted we can test NN



```
Create matrix 1x4 and transpose it to matrix 4x1 at the same time Creating three layers NN by initializing of instance of the class Starting the training process with data above and number of repetitions of 5000  
Showing the output results after training After training process was finished and weights are adjusted we can test NN  
  
input_set_for_training = np.array([[1, 1, 1], [1, 0, 1], [0, 0, 1], [0, 1, 1]])  
output_set_for_training = np.array([[1, 0, 1, 0]]).T  
  
nn = NeuralNetwork()  
  
errors = nn.training_process(input_set_for_training, output_set_for_training, 5000)  
  
print()  
print('Output results after training: ')  
print(nn.layer_2)  
print()  
  
print('Output result for testing data = ', nn.run_nn(np.array([1, 0, 0])))  
  
# fig = plt.figure()  
plt.plot(errors)  
✓ 0.7s
```

نتیجہ

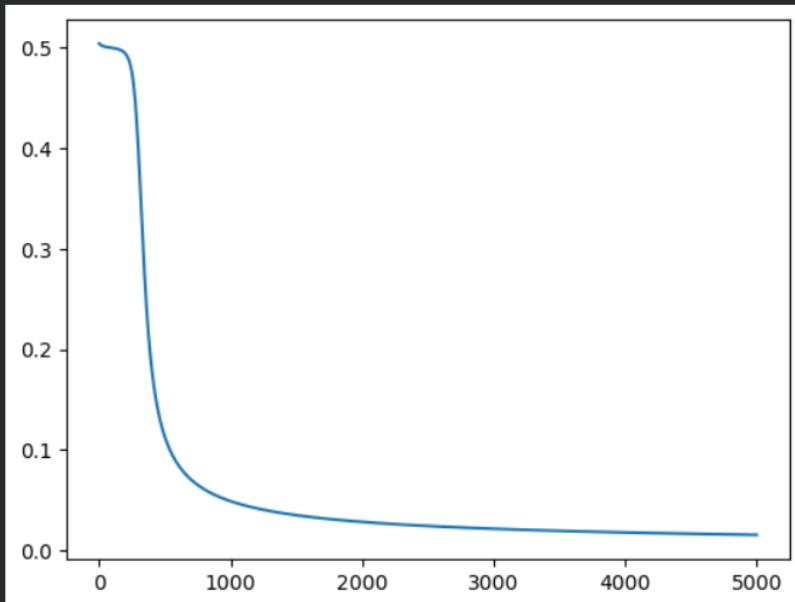
مشاهده می کنیم مدل شبکه‌ی عصبی ما به خوبی ترین شده و فیت شده در طول مرحله‌ها و توانسته داده‌های اموزشی را به خوبی یاد بگیرید و ارروها از ۰,۵ شروع شده و به ۱,۰ رسیده است. و تعمیم دهی شبکه نیز خوب بوده و اورفیت نشده و به مشکلی برخورد نمودیم و می‌توان حتی با درآپ اوت که در سوال‌های دیگر استفاده کردم یا جمع کردن دیتاها بیشتر یا منظم سازی‌ها عملکرد شبکه را بهتر نیز کرد.

```
Final error after 0 epoch = 0.5035899680972746
Final error after 500 epoch = 0.11276292277178596
Final error after 1000 epoch = 0.04919636361412779
Final error after 1500 epoch = 0.03532123858452277
Final error after 2000 epoch = 0.028683045919303228
Final error after 2500 epoch = 0.024637702262591363
Final error after 3000 epoch = 0.021853643867947945
Final error after 3500 epoch = 0.019791311427645494
Final error after 4000 epoch = 0.018186317007655432
Final error after 4500 epoch = 0.016892254711503706

Output results after training:
[[0.98519849]
 [0.01661128]
 [0.986557]
 [0.01843516]]

Output result for testing data = [0.01072646]

[<matplotlib.lines.Line2D at 0x15dfb2921f0>]
```



Using 'seed' for the random generator Modeling three layers Neural Network

Input layer (Layer 0) has three parameters Hidden layer (Layer 1) has four neurons Output layer (Layer 2) has one neuron

Initializing weights between input and hidden layers
The values of the weights are in range from -1 to 1

We receive matrix 3x4 of weights (3 inputs in Layer 0 and 4 neurons in Layer 1)

Initializing weights between hidden and output layers
The values of the weights are in range from -1 to 1

We receive matrix 4x1 of weights (4 neurons in Layer 1 and 1 neuron in Layer 2)

Creating a variable for storing matrix with output results

```
class NeuralNetwork():
    def __init__(self):
        np.random.seed(1)

        self.weights_0_1 = 2 * np.random.random((3, 4)) - 1
        self.weights_1_2 = 2 * np.random.random((4, 1)) - 1

        self.layer_2 = np.array([])
```

Python

Creating function for normalizing weights and other results by Sigmoid curve Creating function for calculating a derivative of Sigmoid function (gradient of Sigmoid curve) Which is going to be used for back propagation - correction of the weights This derivative shows how good is the current weights

+ Code + Markdown

```

Creating function for normalizing weights and other results by Sigmoid curve Creating function for calculating a derivative of Sigmoid function (gradient of Sigmoid curve) Which is going to be used for back propagation - correction of the weights This derivative shows how good is the current weights

def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def derivative_of_sigmoid(self, x):
    return x * (1 - x)

```

Creating function for running and testing NN after training Feed forward through three layers in NN Results are returned in normalized form in appropriate dimensions

```

def run_nn(self, set_of_inputs):
    layer_0 = set_of_inputs
    layer_1 = self.sigmoid(np.dot(layer_0, self.weights_0_1))
    layer_2 = self.sigmoid(np.dot(layer_1, self.weights_1_2))
    return layer_2

```

Feeding our NN with training set and calculating output
Feed forward through three layers in NN
With 'numpy' library and function 'dot' we multiply matrices with values in layers to appropriate weights
Results are returned in normalized form in appropriate dimensions
Creating function for training the NN
Using Backpropagation for calculating values to correct weights
Calculating an error for output layer (Layer 2) which is the difference between desired output and obtained output
We subtract matrix 4x1 of received outputs from matrix 4x1 of desired outputs

```

def training_process(self, set_of_inputs_for_training, set_of_outputs_for_training, epochs):
    errors = []
    for i in range(epochs):

        layer_0 = set_of_inputs_for_training
        layer_1 = self.sigmoid(np.dot(layer_0, self.weights_0_1))
        self.layer_2 = self.sigmoid(np.dot(layer_1, self.weights_1_2))

        layer_2_error = set_of_outputs_for_training - self.layer_2
        errors.append(np.mean(np.abs(layer_2_error)))

if (i % 500) == 0:
    print("Final error after", i, "epoch =", np.mean(np.abs(layer_2_error)))

```

Showing the error each 500 iterations to track the improvements

Calculating delta for output layer (Layer 2) Using sign '*' instead of function 'dot' of numpy library In this way matrix 4x1 will be multiplied by matrix 4x1 element by element
Calculating an error for hidden layer (Layer 1) Multiplying delta_2 by weights between hidden layer and output layer Shows us how much hidden layer (Layer 1) influences on to the layer_2_error
Calculating delta for hidden layer (Layer 1)
Implementing corrections of weights

۶-یک مدل MLP روی دیتاست MNIST طراحی کنید به طوری که به دقت حداقل ۹۵ درصد دست یابد. در گزارش خود توضیح دهید تعداد لایه ها و نورون ها را بر چه اساس انتخاب کردید. در پایان کار با استفاده از matplotlib نمودار توابع Loss و Accuracy را رسم کنید و در گزارش خود بیاورید.

[پاسخ \(۶\)](#)

[تعداد لایه ها :](#)

در ابتدا تعداد کلاس ها و سایز عکسهای ورودی را تعریف میکنیم. سپس دیتای آموزشی و تست را لود میکنیم و تقسیم بر ۲۵۵ میکنیم تا به رنچ ۰ تا ۱ اسکیل شوند.

سپس لیبل داده های تست و ترین را categorical_to_mincnim تا به صورت یک بردار به تعداد کلاس های ما در بیانند و مقدار هر خانه از بردار احتمال مربوط بودن نمونه به آن کلاس میشود که بین صفر تا یک است.

در مرحله‌ی بعد مدل خود را به صورت sequential تعریف میکنیم. در لایه‌های اولیه و میانی از تابع فعالسازی `relu` و در لایه‌ی آخر از تابع فعالسازی softmax استفاده میکنیم. زیرا مسئله‌ی ما کلاس بندی چند کلاسه است. لایه‌ها نیز به صورت کامل متصل هستند. برای بهبود مدل خود از dropout نیز استفاده کردہایم.

- یک لایه حذف یا Dropout با نرخ ۰,۵، که به منظور جلوگیری از بیشبرازش یا overfitting به کار میرود. این لایه به طور تصادفی برخی از نورونهای لایه قبلی را حذف میکند و باعث میشود مدل به ویژگیهای مهمتر توجه کند.

مدل خود را کامپایل و روی داده‌های آموزشی اجرا میکنیم. بدلیل نوع مسئله از تابع ضرر crossentropy_categorical را از تابع بهینه ساز adam استفاده می‌کنیم. سایز batch ها را ۱۲۸ و تعداد ایپاک ها را ۱۵ تعریف میکنیم. سپس نمودار دقت و خطای را رسم میکنیم.

همانطور که میبینیم به دقت بالای ۹۸٪ برای داده‌های آموزشی و validation رسیده و همچنین خطای کمتر از ۳٪ را داریم

دو مدل پیاده سازی مدل :

روش اول پیاده سازی مدل

-همچنین قسمت سکوانشال مدل را بر اساس دانش درس بینایی ماشین کد زدم و از لایه‌های Conv2D و maxpool استفاده کردم اما می‌توانستم از لایه‌ای فولی کانکتد و دنس مانند سوال دیتابست هدا استفاده کرد ولی برای دقت بالاتر از چند لایه متناوب استفاده کردم که اگر پیچیده‌تر کرد شبکه را و هزینه بالا می‌رود ولی دقت بسیار بالا و لاس بسیار کم در ایپوک های نهایی خود مشاهده می‌کنیم . درصد دقت در نهایت خواهیم داشت .

۷ لایه :

یک لایه ورودی : تک کاناله شکل داده $784 = 28 * 28$

یک لایه خروجی : یکی از ۱۰ کلاس ۱۰ نورون

۵ لایه نهان

- یک لایه ورودی که شکل داده‌های ورودی را به عنوان پارامتر می‌گیرد. این لایه هیچ محاسباتی را انجام نمی‌دهد، اما شکل ورودی را برای بقیه شبکه تعریف می‌کند .

• یک لایه کانولوشن که ۳۲ فیلتر به اندازه ۳ در ۳ را با عملکرد فعال سازی `relu` روی ورودی اعمال می‌کند. این لایه با کشیدن فیلترها روی ورودی و اعمال تابع فعال سازی روی نتیجه، ویژگی‌ها را از ورودی استخراج می‌کند.

• یک لایه ادغام حداقل که با گرفتن حداقل مقدار در هر پنجره ۲ در ۲، اندازه ورودی را کاهش می‌دهد. این لایه به کاهش هزینه‌های محاسباتی و جلوگیری از برآش بیش از حد با کاهش نومه ورودی کمک می‌کند.

• یک لایه کانولوشنال دیگر که ۶۴ فیلتر در اندازه ۳ در ۳ را با عملکرد فعال سازی `relu` روی ورودی اعمال می‌کند. این لایه با اعمال فیلترهای بیشتر و افزایش عمق شبکه، ویژگی‌های بیشتری را از ورودی استخراج می‌کند.

• یک لایه max pooling دیگر که با گرفتن حداقل مقدار در هر پنجره ۲ در ۲، اندازه ورودی را کاهش می‌دهد. این لایه همان عملکرد لایه max pooling قبلی را انجام می‌کند.

• یک لایه مسطح که ورودی را به یک بردار یک بعدی تغییر شکل می‌دهد. این لایه با صاف کردن ورودی چند بعدی در یک بردار، ورودی را برای لایه متراکم نهایی آماده می‌کند .

• یک لایه حذفی که به طور تصادفی ۵۰ درصد از واحدهای ورودی را حذف می کند. این لایه با افزودن مقداری نویز و تنظیم به شبکه به جلوگیری از برآش بیش از حد کمک می کند.

• یک لایه متراکم که ۱۰ واحد با عملکرد فعال سازی softmax خروجی می دهد. این لایه وظیفه طبقه بندی نهایی را با نگاشت بردار ورودی به ۱۰ واحد خروجی انجام می دهد که هر یک نشان دهنده یک کلاس است.تابع فعال سازی softmax تضمین می کند که واحدهای خروجی تا ۱ جمع می شوند و می توان آنها را به عنوان احتمال تفسیر کرد.

• یک لایه ورودی که تعداد نورونهای آن برابر با شکل دادهای ورودی است. در این مثال، شکل دادهای ورودی ۲۸ در ۲۸ در ۱ است که نشان میدهد که تصاویر سیاه و سفید با اندازه ۲۸ در ۲۸ پیکسل هستند. بنابراین، تعداد نورونهای لایه ورودی ۷۸۴ است.

• یک لایه پیچشی که تعداد نورونهای آن برابر با تعداد فیلترهای آن است. در این مثال، تعداد فیلترها ۳۲ است که هر کدام از اندازه ۳ در ۳ هستند. بنابراین، تعداد نورونهای لایه پیچشی اول ۳۲ است.

• یک لایه ادغام حداکثر که تعداد نورونهای آن برابر با تعداد نورونهای لایه قبلی است. در این مثال، تعداد نورونهای لایه قبلی ۳۲ است. بنابراین، تعداد نورونهای لایه ادغام حداکثر اول ۳۲ است.

• یک لایه پیچشی دیگر که تعداد نورونهای آن برابر با تعداد فیلترهای آن است. در این مثال، تعداد فیلترها ۶۴ است که هر کدام از اندازه ۳ در ۳ هستند. بنابراین، تعداد نورونهای لایه پیچشی دوم ۶۴ است.

• یک لایه ادغام حداکثر دیگر که تعداد نورونهای آن برابر با تعداد نورونهای لایه قبلی است. در این مثال، تعداد نورونهای لایه قبلی ۶۴ است. بنابراین، تعداد نورونهای لایه ادغام حداکثر دوم ۶۴ است.

• یک لایه صافی که تعداد نورونهای آن برابر با تعداد عناصر لایه قبلی است. در این مثال، تعداد عناصر لایه قبلی ۶۴ در ۴ در ۴ است که برابر با ۱۰۲۴ است. بنابراین، تعداد نورونهای لایه صافی ۱۰۲۴ است.

• یک لایه خروجی که تعداد نورونهای آن برابر با تعداد کلاسهای مورد نظر است. در این مثال، تعداد کلاسها ۱۰ است که نشان میدهد که ده عدد انگلیسی هستند. بنابراین، تعداد نورونهای لایه خروجی ۱۰ است.

دلیل استفاده از رلو

ایه های کانولوشن و یادگیری عمیق: محبوب ترین تابع فعال سازی برای آموزش لایه های کانولوشن و مدل های یادگیری عمیق است. سادگی محاسباتی: تابع یکسو کننده برای پیاده سازی بی اهمیت است و فقط به تابع $\max()$ نیاز دارد. پراکندگی نمایشی: یک مزیت مهم تابع یکسو کننده این است که قادر به خروجی یک مقدار صفر واقعی است. رفتار خطی: بهینه سازی یک شبکه عصبی زمانی که رفتار آن خطی یا نزدیک به خطی باشد آسان تر است

دلیل استفاده از softmax

تابع فعال سازی softmax با آسان تر کردن تفسیر خروجی های شبکه عصبی، این کار را برای شما ساده می کند! تابع فعال سازی softmax خروجی های خام شبکه عصبی را به بردار احتمالات تبدیل می کند، که اساساً یک توزیع احتمال روی کلاس های ورودی است.

دلیل استفاده از آدام

تایج بهینه ساز Adam عموماً بهتر از هر الگوریتم بهینه سازی دیگری است، زمان محاسبات سریع تری دارد و به پارامترهای کمتری برای تنظیم نیاز دارد. به همین دلیل، Adam به عنوان بهینه ساز پیش فرض برای اکثر برنامه ها توصیه می شود.

دلیل استفاده از categorical_crossentropy نتروپی متقطع طبقه ای برای طبقه بندی چند طبقه استفاده می شود. آنتروپی متقطع با واگرایی KL متفاوت است اما می توان آن را با استفاده از واگرایی KL محاسبه کرد. همچنین با از دست دادن گزارش متفاوت است، اما زمانی که به عنوان یکتابع از دست دادن یادگیری ماشین استفاده می شود، همان مقدار را محاسبه می کند.

روش دوم:

: ۷ لایه دارد با استفاده از mlp ساده و بدون کانولوشنی ها ساخته شده تعداد لایه ها و نورون ها با توجه به عوامل مختلف تایین می شود :

- نوع مسئله و داده هایی که قرار است حل شوند. برای مثال، برای دسته بندی تصاویر، معمولاً از شبکه های عصبی پیچشی استفاده می شود که دارای چندین لایه پیچشی و ادغام حداکثر هستند. برای پیش بینی دنباله های زمانی، معمولاً از شبکه های عصبی بازگشتی استفاده می شود که دارای چندین لایه بازگشتی و حافظه کوتاه مدت بلند هستند.
- پیچیدگی و تنوع الگوهای موجود در داده ها. برای مثال، برای شناسایی چهره، معمولاً از شبکه های عصبی عمیق استفاده می شود که دارای چندین لایه پنهان هستند. برای شناسایی دستنوشته ها، معمولاً از شبکه های عصبی ساده تر استفاده می شود که دارای یک یا دو لایه پنهان هستند.
- منابع محاسباتی و حافظه های که در دسترس هستند. برای مثال، برای آموزش شبکه های عصبی بسیار عمیق، معمولاً از کارت های گرافیکی قوی و حافظه های بزرگ استفاده می شود. برای آموزش شبکه های عصبی کم عمیق، معمولاً از پردازنده های مرکزی و حافظه های کمتر استفاده می شود.
- تجربه و دانش فردی که شبکه عصبی را طراحی می کند. برای مثال، برای انتخاب بهترین تابع فعال سازی، تعداد نورون ها، نرخ یادگیری و دیگر پارامترهای شبکه عصبی، معمولاً از دانش نظری و تجربی در زمینه یادگیری ماشین و شبکه های عصبی استفاده می شود.

این پیوت ورودی شکل ۲۸*۲۸

خروجی یکی از ده نورون کلاس

لایه های پنهان : ۵ لایه و تعداد نورون دو لایه اخر ۱۰ و بقیه لایه ها ۱۲۸

dense_18 (Dense)	(None, 128)	100480
flatten_6 (Flatten)	(None, 128)	0
activation_18 (Activation)	(None, 128)	0

dense_19 (Dense)	(None, 128)	16512
activation_19 (Activation)	(None, 128)	0
dense_20 (Dense)	(None, 10)	1290
activation_20 (Activation)	(None, 10)	0

```
model_temp = Sequential()

model_temp.add(layers.Dense(128, input_shape=(x_train.shape[1] * x_train.shape[2],)))
model_temp.add(layers.Flatten())
model_temp.add(layers.Activation('relu'))

model_temp.add(layers.Dense(128))
model_temp.add(layers.Activation('relu'))

model_temp.add(layers.Dense(10))
model_temp.add(layers.Activation('softmax'))

model_temp.summary()

Model: "sequential_4"

Layer (type)                 Output Shape              Param #
=====
```

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 128)	100480
flatten_4 (Flatten)	(None, 128)	0
activation_12 (Activation)	(None, 128)	0
dense_13 (Dense)	(None, 128)	16512
activation_13 (Activation)	(None, 128)	0
dense_14 (Dense)	(None, 10)	1290
activation_14 (Activation)	(None, 10)	0

```
=====
Total params: 118282 (462.04 KB)
Trainable params: 118282 (462.04 KB)
Non-trainable params: 0 (0 KB)
```

```
# Input Layer
model_temp.add(layers.Dense(128, input_shape=(x_train.shape[1] *
x_train.shape[2],)))
model_temp.add(layers.Activation('relu'))
```

```
# Hidden Layer
```

```
model_temp.add(layers.Dense(128))
model_temp.add(layers.Activation('relu'))

# Output Layer
model_temp.add(layers.Dense(10))
model_temp.add(layers.Activation('softmax'))
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense (Dense)	(None, 10)	16010

```
Total params: 34826 (136.04 KB)
```

```
Trainable params: 34826 (136.04 KB)
```

```
Non-trainable params: 0 (0.00 Byte)
```

```
Sequential mode & Model summary
```

```
model = Sequential(  
    [  
        Input(shape=input_shape),  
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  
        layers.MaxPooling2D(pool_size=(2, 2)),  
        layers.Flatten(),  
        layers.Dropout(0.5),  
        layers.Dense(num_classes, activation="softmax"),  
    ]  
)  
  
model.summary()
```

```
Number of total class 0 , 1,2,..,9

from tensorflow.keras import datasets
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,Dense
```

```
[]
```

Load dataset

```
num_classes = 10
input_shape = (28, 28, 1)

# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
1490434/11490434 [=====] - 0s 0us/step
```

```
Convert to categorical
```

```
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

```
Sequential mode & Model summary
```

```
model = Sequential(
    [
        Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense (Dense)	(None, 10)	16010

```
Total params: 34826 (136.04 KB)
```

```
Trainable params: 34826 (136.04 KB)
```

```
Non-trainable params: 0 (0.00 Byte)
```

```
Compile model & Train model
```

```
batch_size = 128  
epochs = 15
```

```
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

```
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
```

```

Epoch 1/15
422/422 [=====] - 42s 96ms/step - loss: 0.3588 - accuracy: 0.8910 - val_loss: 0.0840 - val_accuracy:
Epoch 2/15
422/422 [=====] - 41s 96ms/step - loss: 0.1139 - accuracy: 0.9652 - val_loss: 0.0577 - val_accuracy:
Epoch 3/15
422/422 [=====] - 40s 94ms/step - loss: 0.0837 - accuracy: 0.9741 - val_loss: 0.0496 - val_accuracy:
Epoch 4/15
422/422 [=====] - 40s 94ms/step - loss: 0.0687 - accuracy: 0.9789 - val_loss: 0.0424 - val_accuracy:
Epoch 5/15
422/422 [=====] - 39s 93ms/step - loss: 0.0595 - accuracy: 0.9812 - val_loss: 0.0403 - val_accuracy:
Epoch 6/15
422/422 [=====] - 39s 93ms/step - loss: 0.0554 - accuracy: 0.9826 - val_loss: 0.0338 - val_accuracy:
Epoch 7/15
422/422 [=====] - 39s 92ms/step - loss: 0.0504 - accuracy: 0.9838 - val_loss: 0.0354 - val_accuracy:
Epoch 8/15
422/422 [=====] - 40s 95ms/step - loss: 0.0463 - accuracy: 0.9853 - val_loss: 0.0363 - val_accuracy:
Epoch 9/15
422/422 [=====] - 41s 98ms/step - loss: 0.0416 - accuracy: 0.9866 - val_loss: 0.0326 - val_accuracy:
Epoch 10/15
422/422 [=====] - 39s 92ms/step - loss: 0.0407 - accuracy: 0.9871 - val_loss: 0.0306 - val_accuracy:
Epoch 11/15
422/422 [=====] - 39s 91ms/step - loss: 0.0385 - accuracy: 0.9877 - val_loss: 0.0307 - val_accuracy:
Epoch 12/15
422/422 [=====] - 40s 94ms/step - loss: 0.0381 - accuracy: 0.9879 - val_loss: 0.0301 - val_accuracy:
Epoch 13/15
...
Epoch 14/15
422/422 [=====] - 39s 92ms/step - loss: 0.0335 - accuracy: 0.9897 - val_loss: 0.0284 - val_accuracy:
Epoch 15/15
422/422 [=====] - 40s 96ms/step - loss: 0.0310 - accuracy: 0.9900 - val_loss: 0.0336 - val_accuracy:

```

Plot accuracy & Plot loss

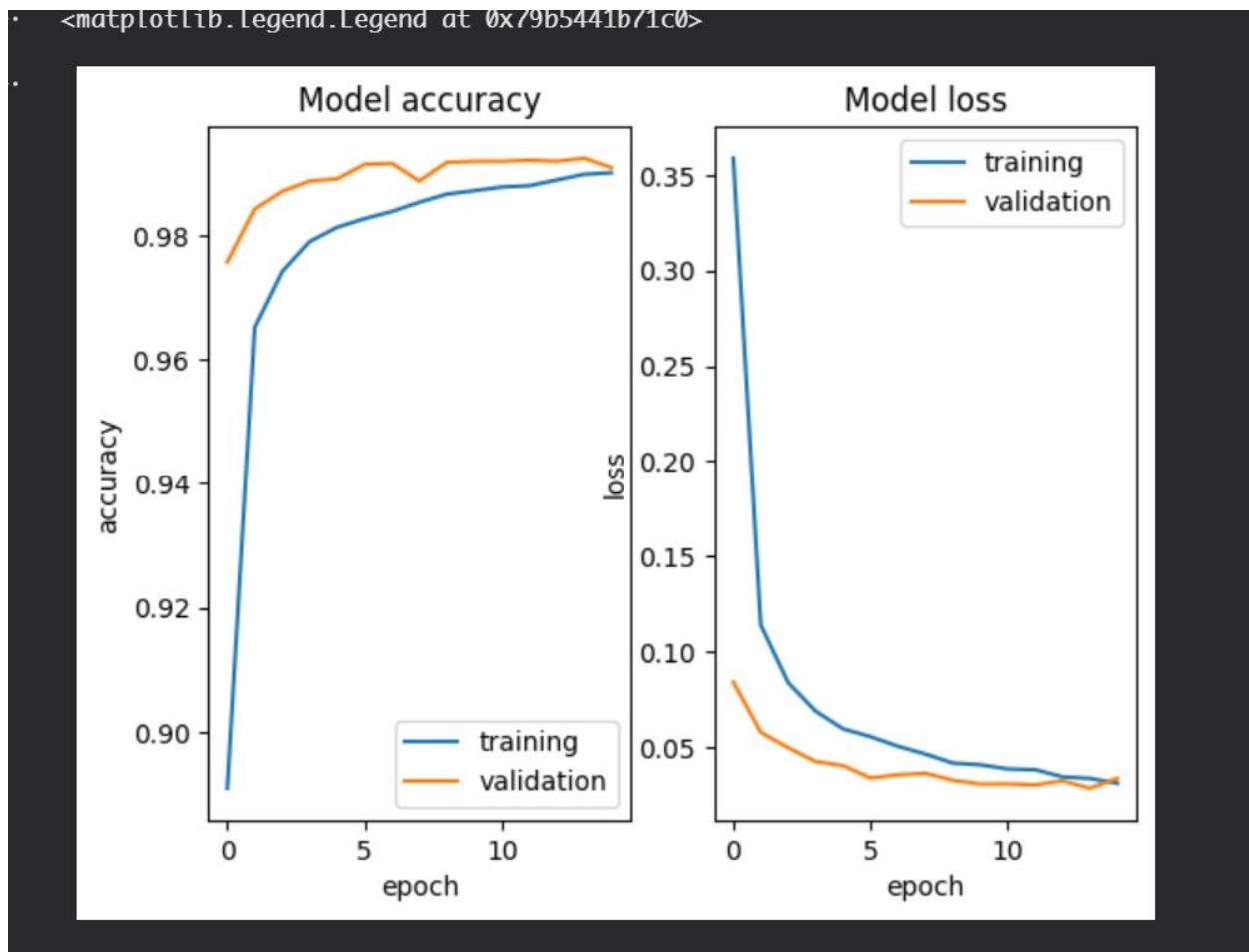
```

fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(history.history['accuracy'])
ax1.plot(history.history['val_accuracy'])
ax1.set_title('Model accuracy')
ax1.set_ylabel('accuracy')
ax1.set_xlabel('epoch')
ax1.legend(['training', 'validation'], loc='lower right')

ax2.plot(history.history['loss'])
ax2.plot(history.history['val_loss'])
ax2.set_title('Model loss')
ax2.set_ylabel('loss')
ax2.set_xlabel('epoch')
ax2.legend(['training', 'validation'], loc='upper right')

```

درستی و صحت مدل ها را از نمودار ها می توان متوجه شد مقدار لاس و دقق در داده های تست و اموزش یک فلوی یکسان را طی می کند و دقق زیادو لاس کم شده



مراجع

سؤال ١ :

<https://studyglance.in/nn/display.php?tno=5&topic=Adaptive-Linear-Neuron>

سؤال ٢ :

[neural network - What is the difference between Perceptron and ADALINE? - Data Science Stack Exchange](#)

<https://www.slideserve.com/walda/newton-raphson-method>

<https://www.slideserve.com/walda/newton-raphson-method>

وبسایت فرادرس

Bing

سؤال ٥ :

[Backpropagation in Neural Network \(NN\) with Python - Valentyn Sichkar \(sichkar-valentyn.github.io\)](#)

سؤال ٦ :

[Simple MNIST convnet \(keras.io\)](#)
