

تمرین دوم هوش محاسباتی

ملیکا محمدی فخار

۹۹۵۲۲۰۸۶

در انجام این تمرین علاوه بر دانسته‌های این جانب از اسلایدهای درسی و <https://chat.openai.com> نیز به عنوان منبع استفاده شده است.

۱

				Year:	Month:	Date:
1) NAND Function For a ADALINE neuron.						
$\alpha = 0, 1$	$b = 1$			w_1, w_r, b		
x_1	x_r	bias	target	random initial weights: $w_1 = 0.1, w_r = 0.1, b = 1$		
1	1	1	-1			
1	-1	1	1			
-1	1	1	1			
-1	-1	1	1			
				$y = \alpha$		
① input = 1, 1 output = -1				$w_{\text{new}} = w_{\text{old}} + \eta(d-y)x_i$ $y = 0.1 + 0.1x_1 + 0.1x_r = 0.7$ $b_{\text{new}} = b_{\text{old}} + \eta(d-y)$ $b = 1 + 0.1(-1-0.7) = 0.15$ $w_1 = 0.1 + 0.1(-1-0.7) = 0.15$ $w_r = 0.1 + 0.1(-1-0.7) = 0.15$		
② input = 1, -1 output = 1				$y = 0.15 + 0.15x_1 + 0.15x_r - 1 = 0.95$ $b = 0.15 + 0.1(1-0.95) = 0.15$ $w_1 = 0.15 + 0.1(1-0.95)x_1 = 0.15$ $w_r = 0.15 + 0.1(1-0.95)x_r = 0.15$		
③ input = -1, 1 output = 1				$y = 0.15 + 0.15x_1 + 0.15x_r - 1 = 0.75$ $b = 0.15 + 0.1(1-0.75) = 0.15$ $w_1 = 0.15 + 0.1(1-0.75)x_1 = 0.15$ $w_r = 0.15 + 0.1(1-0.75)x_r = 0.15$		
④ input = -1, -1 output = 1				$y = 0.15 - 0.15 - 0.15 = 0.25$ $w_1 = 0.15 - 0.1(1-0.25) = 0.14$ $w_r = 0.15 - 0.1(1-0.25) = 0.14$ $((\text{نحوی})) w_1 = 0.15 - 0.1(1-0.25) = 0.14$		

.۲
الف.

تابع فعال‌سازی خطی مقادیر ورودی را به صورت خطی به خروجی نگاشت می‌کند. مانند تابع همانی. این به این معناست که خروجی یک تابع فعال‌سازی خطی مجموع حاصل ضرب ورودی در وزن‌های نظیر هر ورودی است. به لحاظ محاسباتی اگر چندین لایه خطی پشت سر هم داشته باشیم، می‌توان آن‌ها را به یک لایه خطی تبدیل کرد. این توابع ممکن است برای برخی مسائل ابتدایی و ساده مانند رگرسیون خطی مناسب باشند اما توانایی حل مسائل پیچیده‌تر را ندارند.

تابع فعال‌سازی غیرخطی مقادیر ورودی را به صورت غیرخطی به خروجی نگاشت می‌کند و محاسبات پیچیده‌تری دارد. این بدان معناست که دیگر خروجی، مجموع حاصل ضرب ورودی در وزن‌های نظیر هر ورودی نیست. استفاده از توابع غیرخطی در یک شبکه عصبی، به شبکه کمک می‌کند توانایی یادگیری بالاتری داشته باشد و الگوهای پیچیده‌تری را از داده‌های ورودی استخراج کند. از پرکاربردترین توابع فعال‌سازی غیرخطی می‌توان به LU Sigmoid, TanH, ReLU اشاره کرد.

.ب.

اگر مدل را به گونه‌ای طراحی کنیم که وزن‌های اولیه صفر و بایاس رندوم باشد، دیگر مدل قادر به یادگیری نیست. زیرا داده ورودی هرچه باشد ابتدا در وزن‌ها که صفر هستند ضرب شده و حاصل صفر می‌شود، سپس این مقدار با بایاس جمع می‌شود و مقدار خروجی هر نورون برابر با بایاس می‌شود. از آنجا که گرادیان نسبت به وزن‌ها هم صفر خواهد بود، وزن‌ها هرگز تغییر نمی‌کنند و مدل نمی‌تواند در فرایند back propagation نیز وزن‌ها را آپدیت کند.

اگر مدل را به گونه‌ای طراحی کنیم که وزن‌های اولیه رندوم و بایاس صفر باشد، مدل دارای پتانسیل یادگیری خواهد بود اما یادگیری آن زمان برخواهد بود و به کندی همگرا می‌شود. زیرا نورون‌های بدون بایاس ابتدا خروجی‌های کوچک تولید می‌کنند و حتی اگر حاصل ضرب ورودی در وزن صفر شود، آن نورون غیرفعال می‌شود. از این رو بهتر است هیچ‌یک (وزن‌ها و بایاس) را صفر در نظر نگیریم.

.ج.

مدلهایی که تا کنون شناخته‌ایم شامل MLP, Adaline, MADaline, Perceptron می‌باشند. در این میان تصمیم‌گیری‌های ساده هستند و اگر مسئله به گونه‌ای باشد که نیاز به درک الگوهای پیچیده‌تری داده‌ها داشته باشیم، این مدل‌ها خوب عمل نمی‌کنند و قدرت تعمیمدهی پایینی دارند. اگر بخواهیم رتبه‌بندی کنیم، Adaline از MADaline و Perceptron از Adaline بهتر عمل می‌کند.

مدل MLP از آنجا که تعداد لایه‌ها و نورون‌های هر لایه‌اش قابل تنظیم است و می‌تواند از توابع غیرخطی استفاده کند، با تشکیل یک مدل عمیق با توابع فعال‌سازی مناسب، قادر به یادگیری الگوهای پیچیده‌تری در داده‌های است و بنابراین توانایی تعمیم‌پذیری بیشتری نسبت به سه مدل قبلی دارد.

د. روش‌های وفقی

- انتخاب مقدار درست برای نرخ آموزش η بسیار مهم و البته مشکل است
- آیا می‌توان η را به صورت خودکار انتخاب کرد یا اصلاً به آن نیاز نداشت؟
- می‌توان از انحنای منحنی (curvature) هم استفاده کرد
- روش نیوتون: در بسط تیلور از مرتبه دوم هم استفاده کنیم

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^T \nabla f(\mathbf{x}) + \frac{1}{2} \boldsymbol{\epsilon}^T \nabla^2 f(\mathbf{x}) \boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3)$$

- ماتریس Hessian تابع f با ابعاد $d \times d$ به صورت $d \times d H \triangleq \nabla^2 f(\mathbf{x})$ نمایش داده می‌شود
- برای d ‌های بزرگ از لحاظ مصرف حافظه و محاسبات سنگین خواهد بود
- تقریب مرتبه ۲ دارای یک بهینه سراسری و قابل محاسبه است

روش نیوتن

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^T \nabla f(\mathbf{x}) + \frac{1}{2} \boldsymbol{\epsilon}^T \mathbf{H} \boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3)$$

• برای یافتن مقدار بهینه این تابع تقریبی، می‌توان مشتق آن نسبت به $\boldsymbol{\epsilon}$ را برابر با صفر قرار داد

$$\nabla f(\mathbf{x}) + \mathbf{H} \boldsymbol{\epsilon} = 0 \Rightarrow \boldsymbol{\epsilon} = -\mathbf{H}^{-1} \nabla f(\mathbf{x})$$

• بجای نرخ آموزش از معکوس ماتریس Hessian استفاده شده است

$$f(x) = x^2 - 2x + 3 \Rightarrow f'(x) = 2x - 2 \Rightarrow f''(x) = 2$$

• مثال:

$$x_0 = 12 \Rightarrow x \leftarrow x + \boldsymbol{\epsilon} = 12 - \frac{1}{2} \times 22 = 1$$

• در این مثال خاص در یک تکرار از هر نقطه شروعی به جواب بهینه می‌رسد

معادله داده شده در سوال نیز مربوط به استفاده از معکوس تابع Hessian در آپدیت کردن وزن‌هاست.

مزایا:

همگرایی سریعتر: ماتریس هسی اطلاعاتی در مورد منحنی سطح خط ارائه می‌دهد که به بهود همگرایی و سرعت همگرایی در صورت وجود منحنی‌های خط با رفتار مناسب کمک می‌کند. این بهینه‌سازی می‌تواند بهینه‌تر و با کمترین تعداد تکرارها به نقطه کمینه برسد.

بهبود نرخ‌های یادگیری: می‌تواند نرخ‌های یادگیری را برای هر پارامتر تطبیق دهد که مخصوصاً در بهود همگرایی در برخی مناظر خط پیچیده و غیرقابل‌کنترل مفید است.

پایداری بهتر: ماتریس هسی به دلیل کمتر بودن امکان سرزنش و نوسان، می‌تواند فرآیند آموزش را پایدارتر کند و از اختلافات زیاد در نقاط کمینه جلوگیری کند.

مدیریت بهتر مسائل با وضوح ضعیف: در مواردی که تابع خط ویژگی‌های ضعیفتر و قوی‌تر داشته باشد (که در یادگیری عمیق اغلب روی می‌دهد)، ماتریس هسی ممکن است نسبت به متهای اولین مرتبه مقاومت بیشتری داشته باشد.

معایب:

هزینه محاسباتی زیاد: محاسبه و معکوس کردن ماتریس هسی زمانبر است، به ویژه برای شبکهای عصبی بزرگ با تعداد پارامترهای زیاد. این می‌تواند این متد را در عمل برای یادگیری در شبکهای عمیق، غیرممکن کند.

صرف حافظه بالا: ذخیره و انجام عملیات روی ماتریس هسی نیازمند حافظه زیادی است که ممکن است در سختافزارهای معمولی، به ویژه برای شبکهای عمیق، در دسترس نباشد.

حساسیت به نویز: ماتریس هسی حساس به گراییانهای نویزی است که در عمل معمول است. گراییانهای نویزی ممکن است به عدم دقیقت در ماتریس هسی منجر شود و مشکلات همگرایی ایجاد کند.

کاربرد محدود: معمولاً ماتریس هسی در مسائلی با منحنی‌های خط صاف یا کمی مقعر و کمینه‌سازی مسائل اصولی مورد استفاده قرار می‌گیرد. مسائل یادگیری عمیق اغلب مناظر خط غیرمقعر و صعبالعبور دارند و ممکن است ماتریس هسی در مقایسه با متهای اولین مرتبه مزیت معنیداری نداشته باشد.

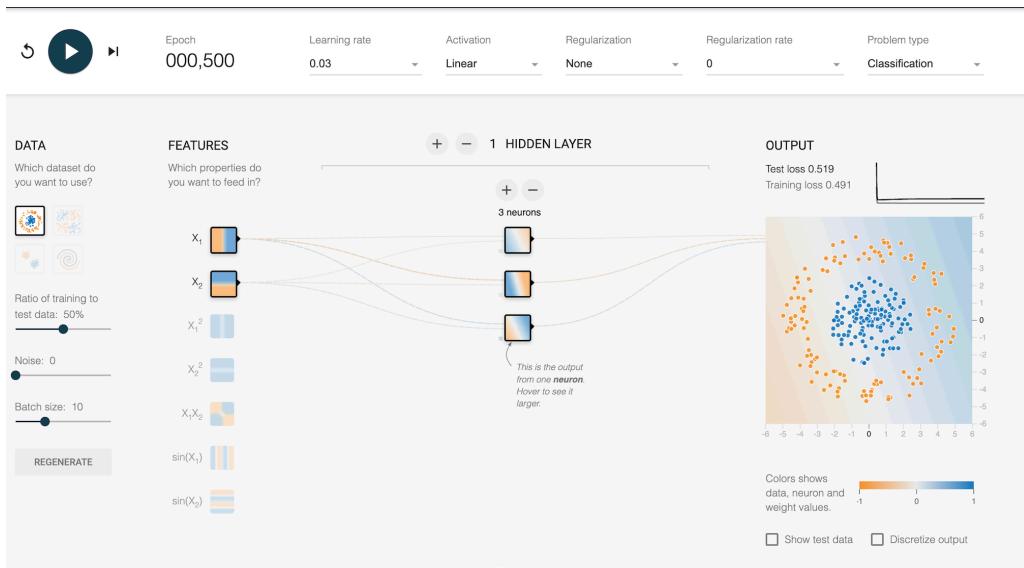
کهوانی در عملیات: ماتریس هسی ممکن است به جزئیات دقیق سطح خط تمرکز کند که ممکن است به بهود تعیین‌پذیری منجر نشود و مشکلات بیشبرازش (overfitting) ایجاد کند.

دیتاست با توزیع دایره:

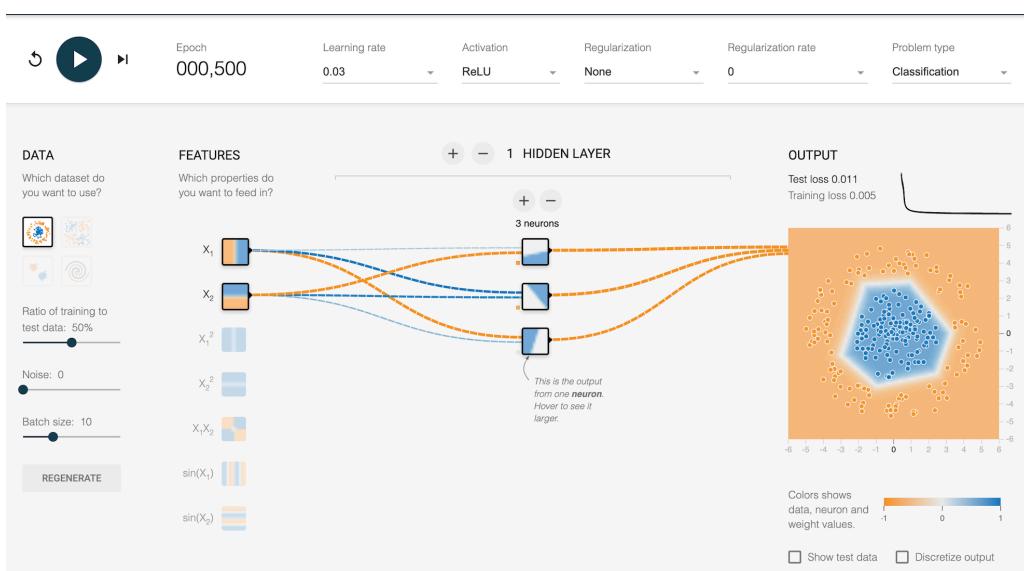
همان‌طور که از شکل داخل تصویر مشخص است، داده‌ها به صورت دو دایره درون یکدیگر قرار گرفته‌اند و مدل بایست معادله یک دایره یا چندضلعی را یاد بگیرد تا بتواند دو دایره را از یکدیگر تفکیک کند.

کارایی عملکرد توابع به ترتیب: TanH, ReLU, Sigmoid, Linear.

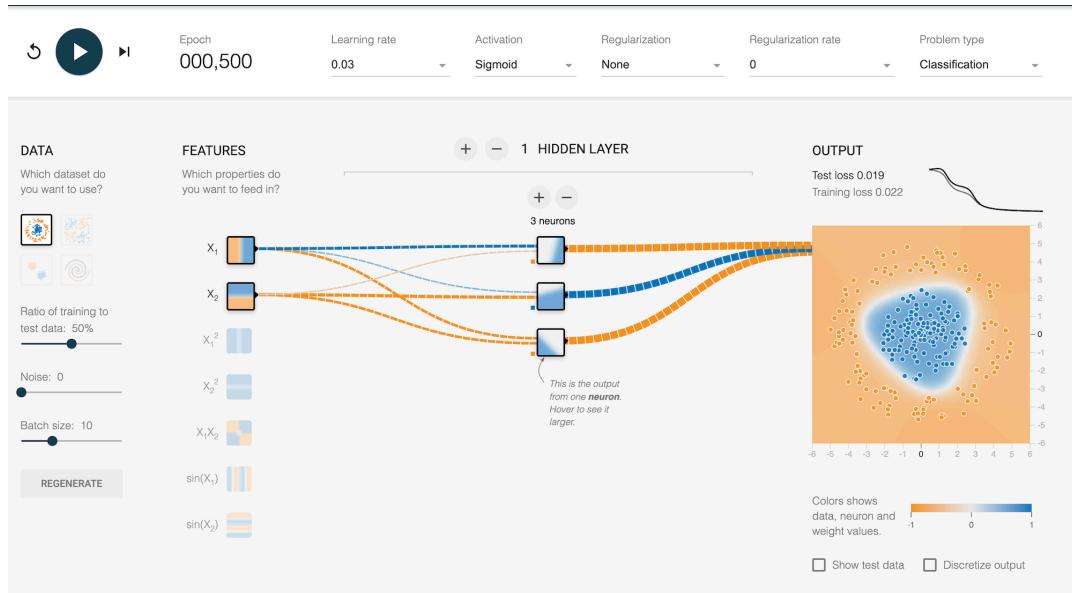
تابع linear: واضح است که به وسیله یک خط نمی‌توان دو دایره را از یکدیگر جدا کرد بنابراین تابع خطی بر روی این مسئله عملکرد مناسبی ندارد و ضرر آن در حدود ۰.۵ است. زیرا در یک مسئله دو کلاسه اگر به صورت رندوم نیز کلاس‌ها را پیش‌بینی می‌کردیم نقت در همین حدود بود.



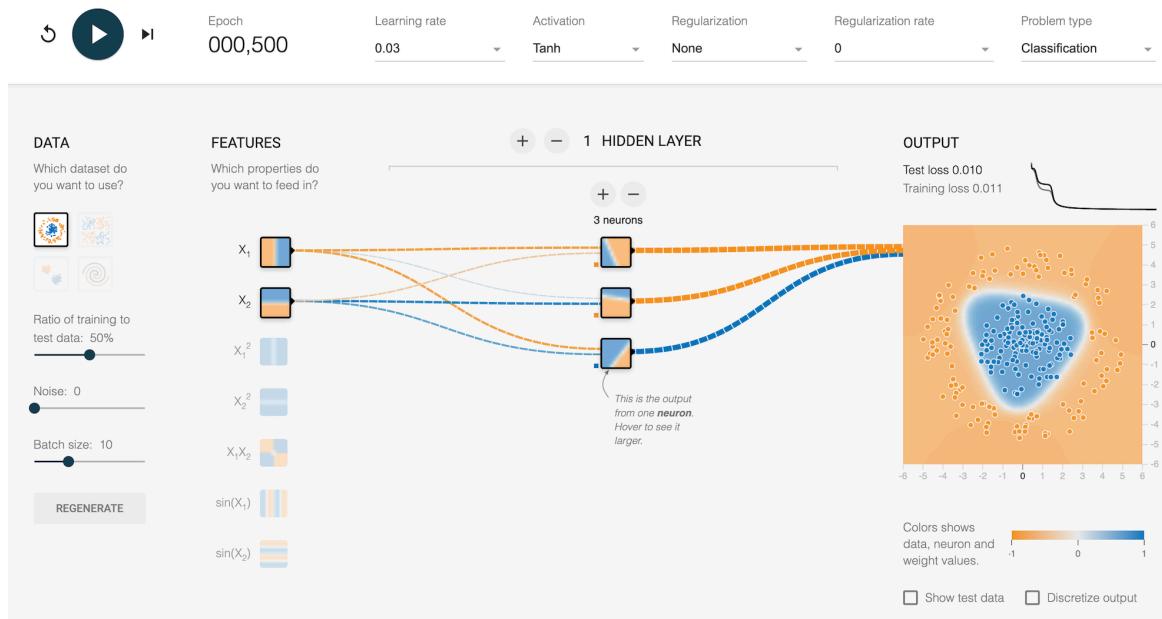
تابع ReLU: این تابع با یادگیری یک مرز به شکل شش ضلعی عملکرد بسیار خوبی در تفکیک این دو کلاس با مقدار ضرر ۰.۱ داشته است.



تابع Sigmoid: میزان خسر با استفاده از این تابع به 0.19 رسیده که نمایانگر عملکرد مناسب این تابع در تفکیک دادها با پیدا کردن یک مرز شبیه به مثلث میباشد.



تابع TanH: میزان خسر این تابع به 0.01 رسیده است که بهترین عملکرد میان توابع فعالسازی ذکر شده را نشان میدهد. این تابع نسبت به Sigmoid سریع‌تر همگرا شده است که یکی از دلایل آن میتواند Zero-centered بودن تابع TanH باشد.

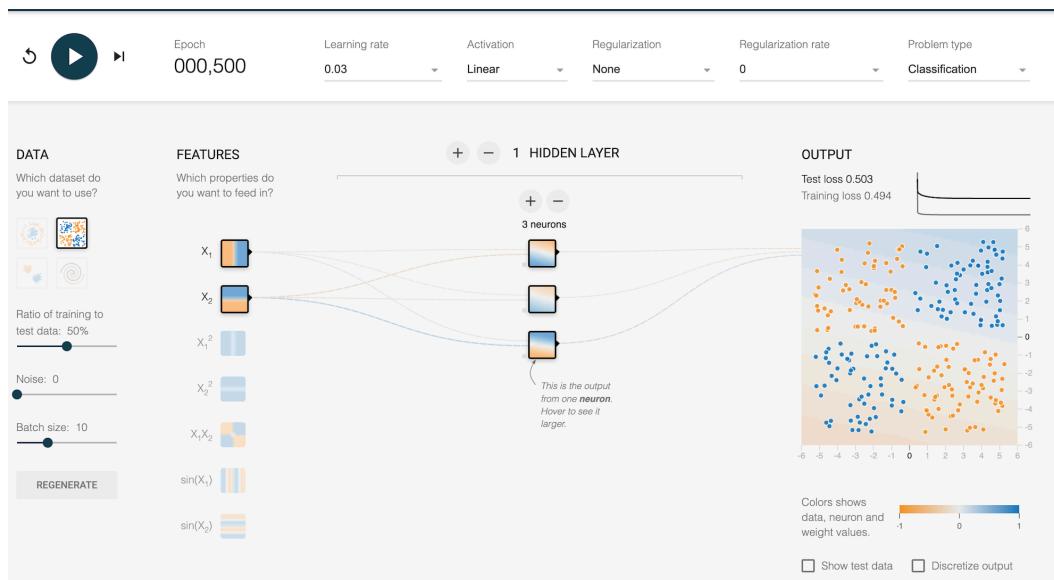


دیتاست با توزیع XOR

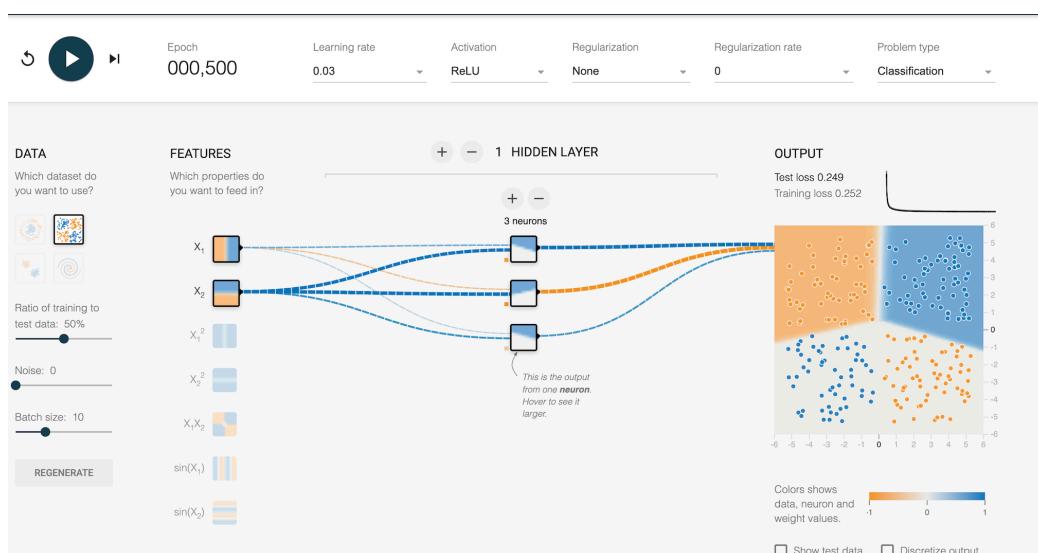
همانطور که در شکل داده مشخص است در این نوع از داده، فضای چهار بخش تقسیم شده است و در هر بخش داده‌ها به صورتی قرار گرفته‌اند که هیچ دو بخش مجاوری داده‌های یکسان ندارد. موقع داریم توابع غیرخطی بتوانند عملکرد نسبتاً خوبی ارائه دهند.

کارایی عملکرد توابع به ترتیب:

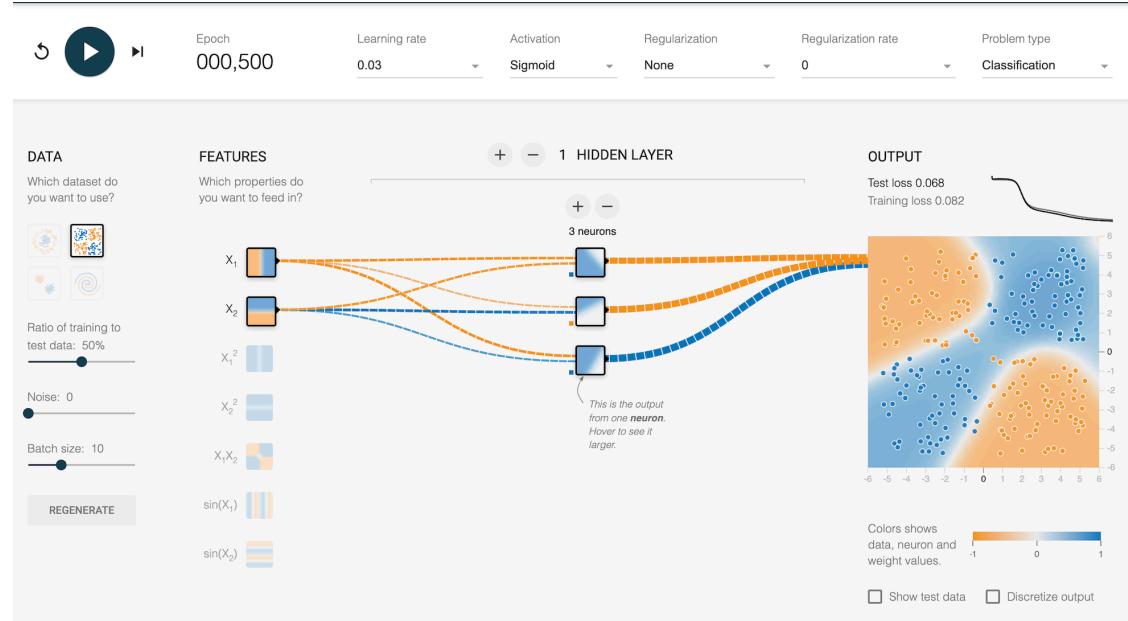
تابع linear: واضح است که به وسیله یک خط نمی‌توان چهار ناحیه را از یکی‌گر جدا کرد بنابراین تابع خطی بر روی این مسئله عملکرد مناسبی ندارد و ضرر آن در حدود ۰.۵ است. زیرا در یک مسئله دو کلاسه اگر به صورت رندوم نیز کلاس‌ها را پیش‌بینی می‌کردیم دقต در همین حدود بود.



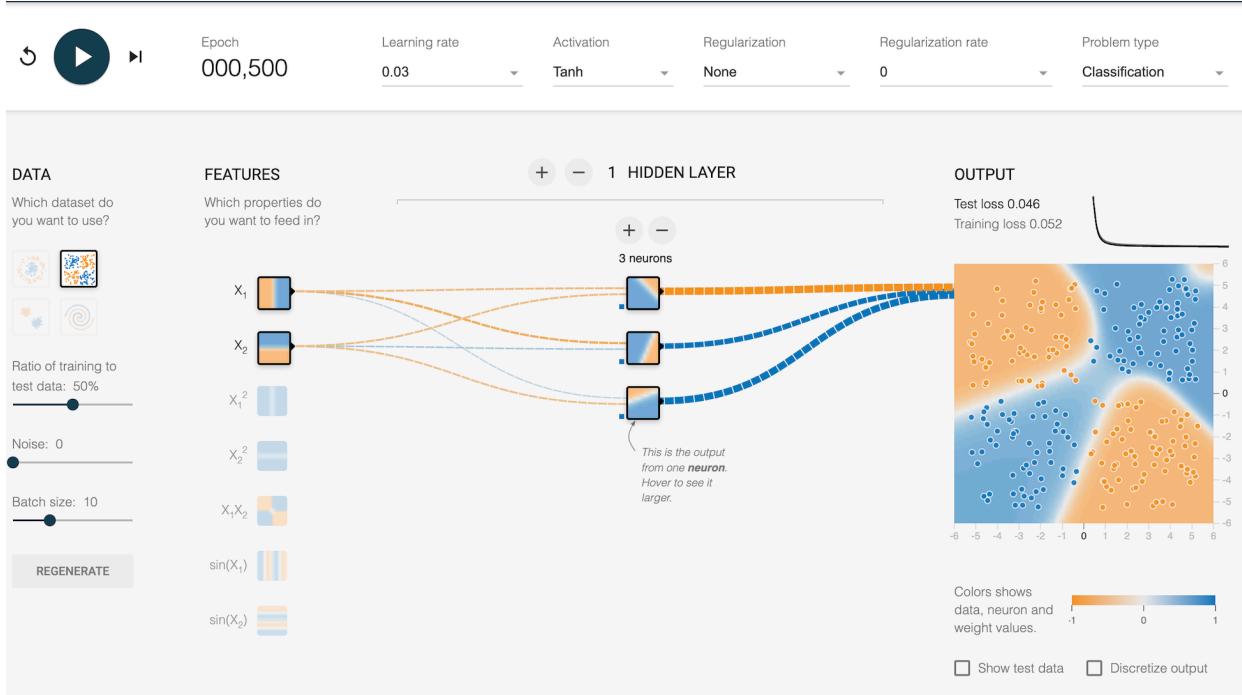
تابع ReLU: این تابع توانسته ۴ ناحیه را به خوبی تفکیک کند و میزان ضرر خود را به ۰.۲۵ برساند و بهترین عملکرد میان توابع بررسی شده بر روی دیتاست XOR را داشته باشد.



تابع Sigmoid: میزان خسر به 0.68 رسیده که عملکرد متوسط مایل به خوبی است و تا حد زیادی تقسیم نواحی صورت پذیرفته است.



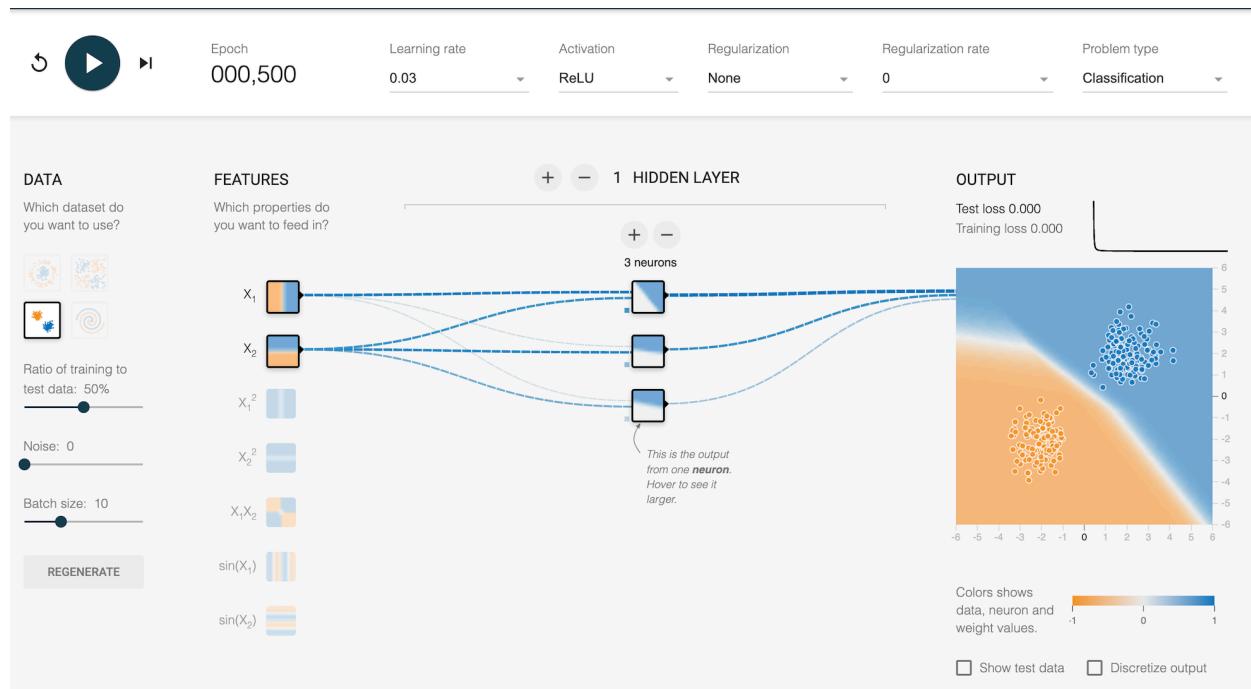
تابع TanH: میزان خسر این تابع به 0.46 رسیده است که عملکرد نسبتاً مناسبی است و مرزها به خوبی تشخیص داده شده‌اند. این تابع نسبت به Sigmoid سریع‌تر همگرا شده است که یکی از دلایل آن میتواند Zero-centered باشد. TanH.



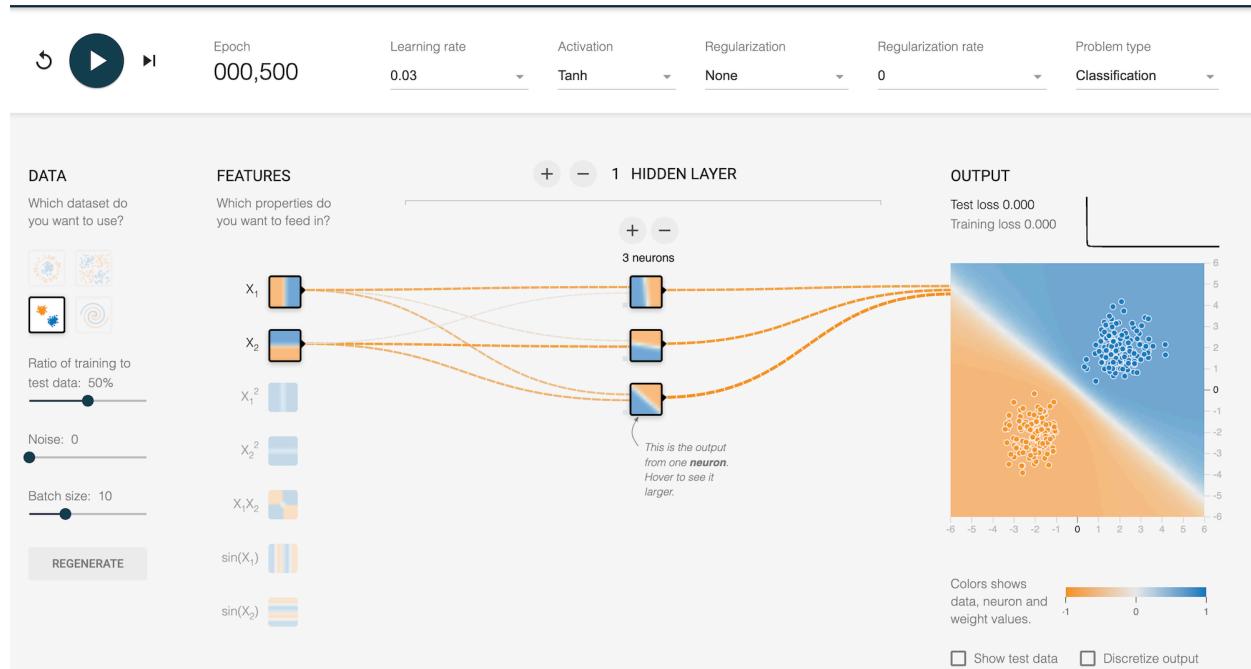
دیتاست با توزیع خطی:

این دیتا ست بسیار ساده است و داده‌ها کاملا از یکدیگر جدا هستند و حتی به وسیله یک خط نیز می‌توان آنها را به خوبی تقسیم کرد. همان‌طور که پیش‌بینی می‌شود ضرر هر 4 تابع به 0 رسیده است و بر روی داده‌های تست نیز تعیین‌پذیری خوبی را به نمایش گذاشت‌اند.

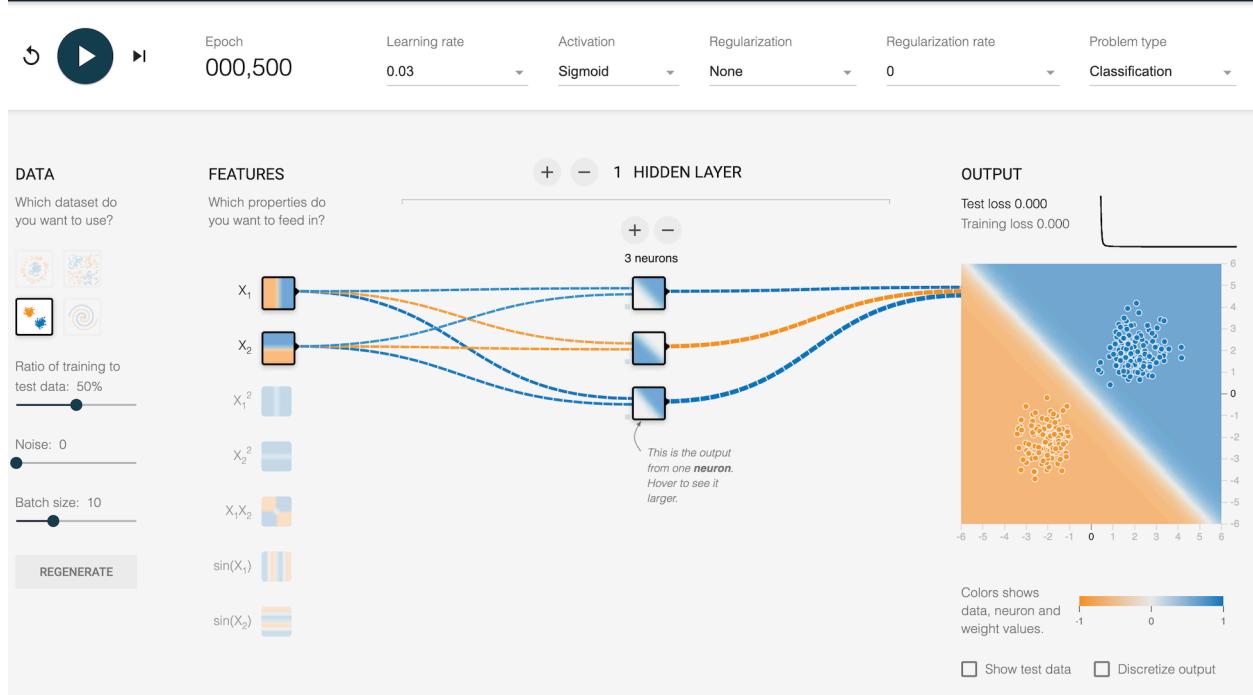
تابع :ReLU



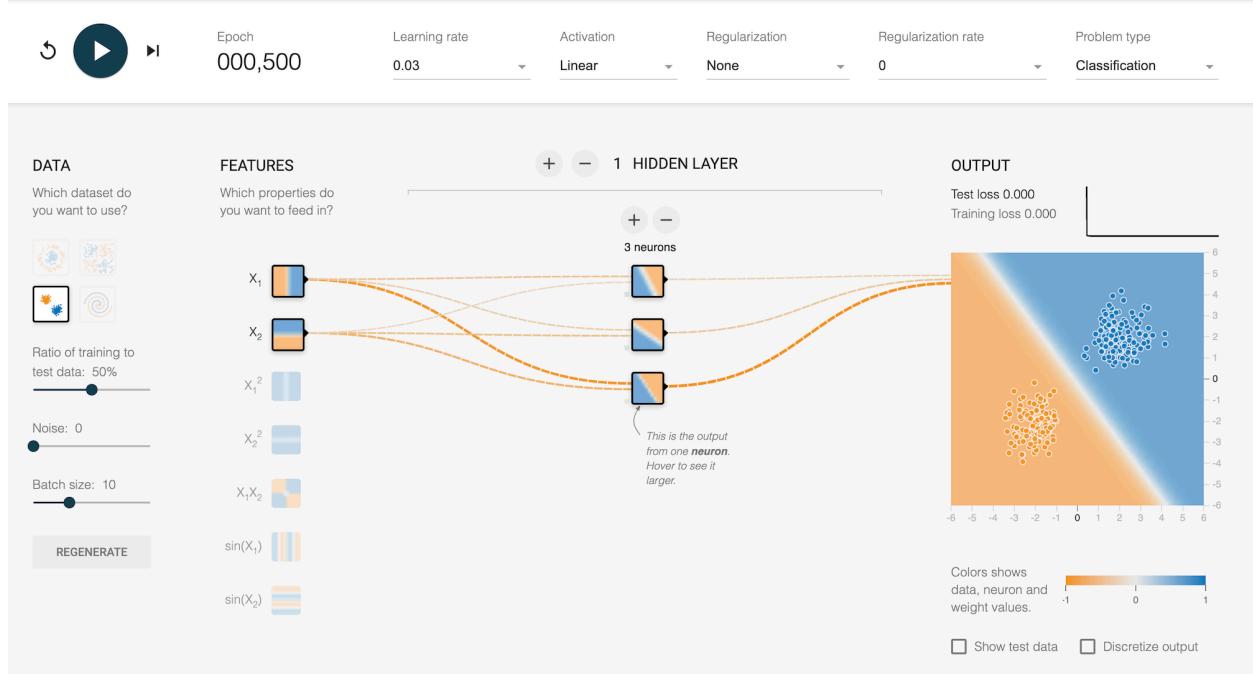
تابع :TanH



تابع Sigmoid



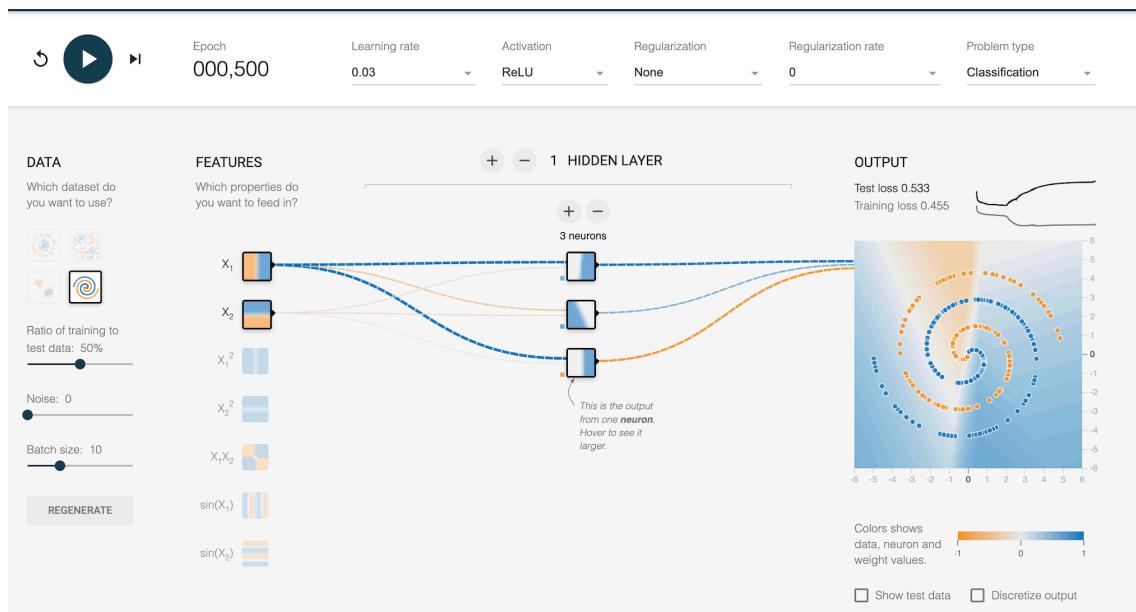
تابع Linear



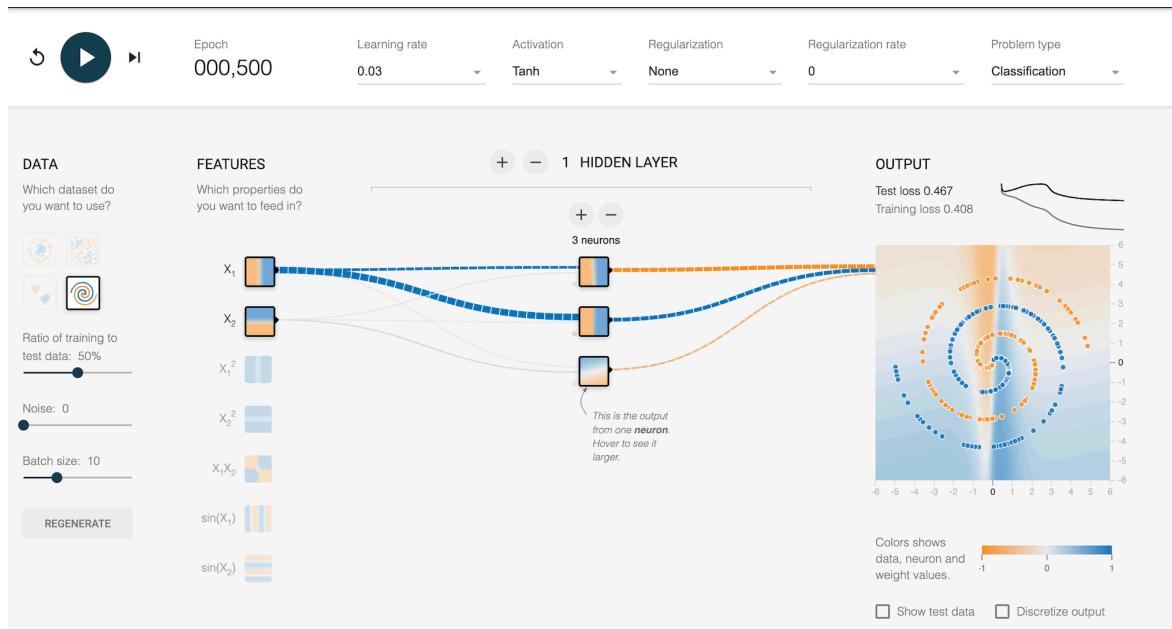
دیتاست با توزیع spiral

این دیتا است بسیار پیچیده است و داده‌ها کاملا در هم تنیده هستند و همانطور که از نتایج مشخص است هیچ یک از توابع فعالسازی نتوانسته‌اند عملکرد مطلوبی بر روی این توزیع داشته باشند. مقدار ضرر این توابع در حدود ۰.۵ است که این بسیار بد است زیرا عملکرد یک تابع رندوم در مسئله طبقه‌بندی دو کلاسه نیز در همین حدود است. بنابراین برای دستیابی به نتیجه مطلوب بر روی این تابع لازم است شبکه پیچیده‌تر و عمیق‌تر باشد تا بتواند الگوهای پیچیده‌تری استخراج کند. کارایی عملکرد توابع به ترتیب:

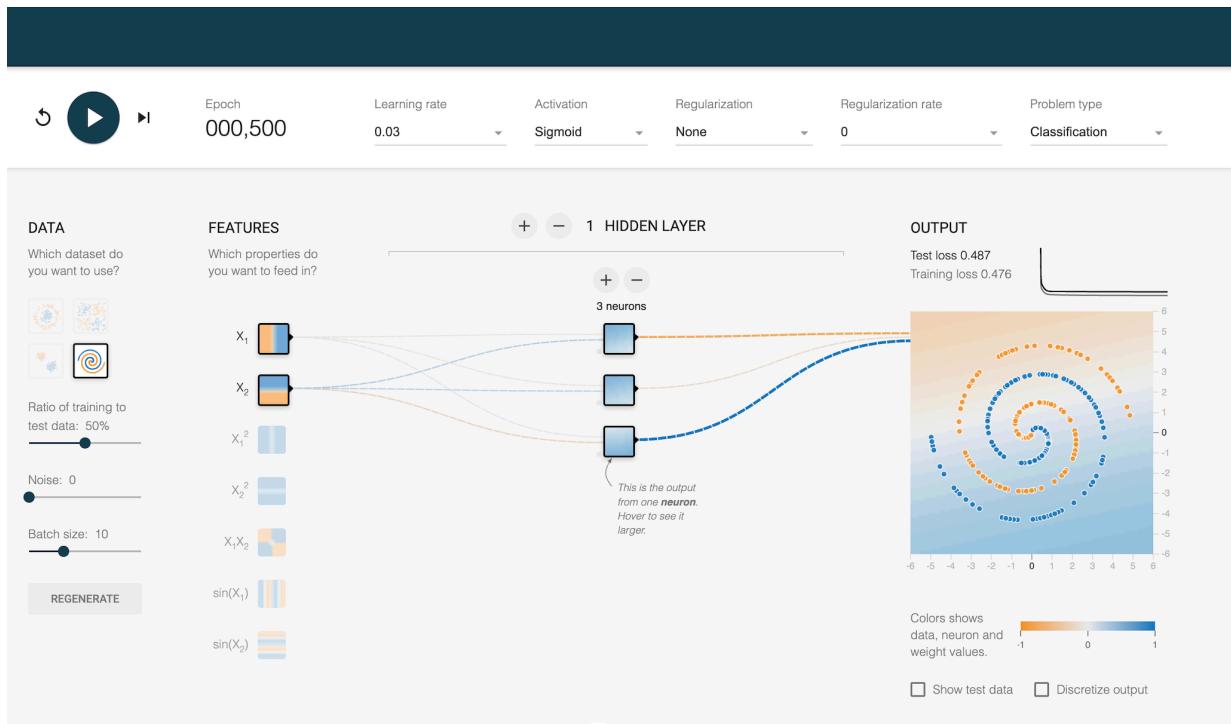
تابع :ReLU



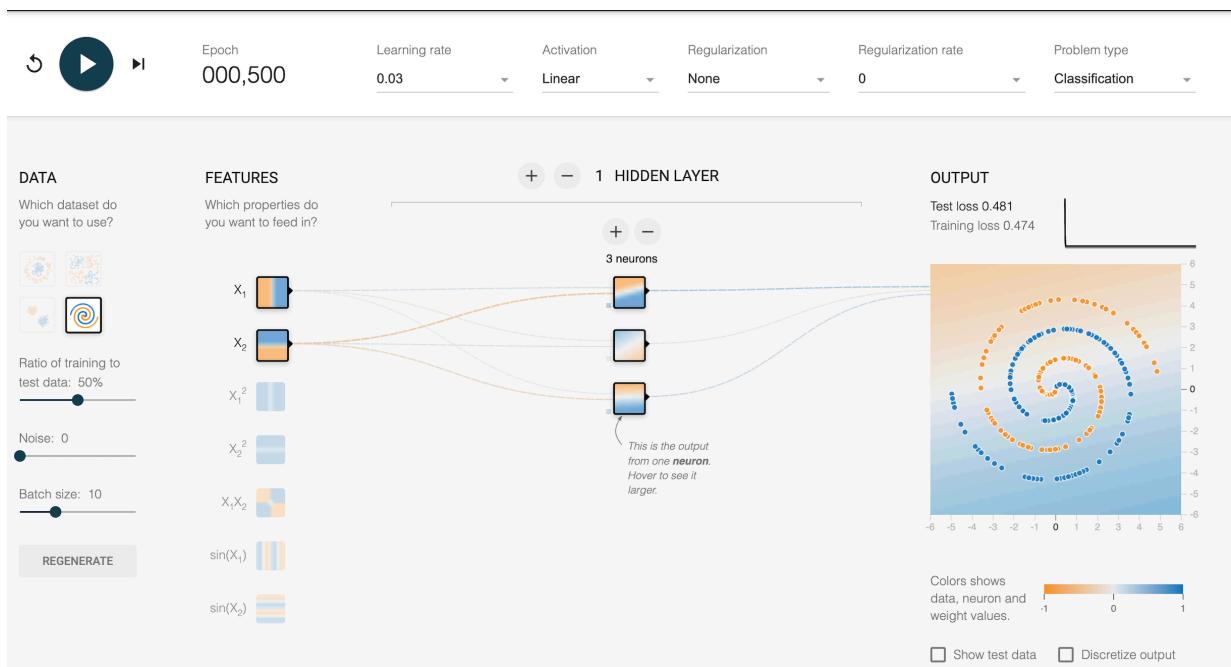
تابع :TanH



:تابع



:تابع



۴. فایل تکمیل شده Q2_4 ضمیمه شده است. ابتدا به توضیح کدهای اضافه شده میپردازیم:
در این بخش مدل از نوع `sequential` تعریف شده است که میتوان لایه‌ها را یکی یکی به آن اضافه کرد. لایه‌ها از نوع `dense` انتخاب شده‌اند که دارای ۱۲۸، ۶۴ و ۱۰ لایه هستند (به نظر میرسد مسئله از نوع طبقبندی است). توابع فعالسازی دو لایه `hidden` از نوع `relu` و برای لایه آخر از `softmax` استفاده شده است.

```
model = Sequential()
model.add(Dense(128, input_dim=25))
model.add(Activation('relu'))
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
dense (Dense)	(None, 128)	3328
activation (Activation)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8256
activation_1 (Activation)	(None, 64)	0
dense_2 (Dense)	(None, 10)	650
activation_2 (Activation)	(None, 10)	0
<hr/>		
Total params: 12234 (47.79 KB)		
Trainable params: 12234 (47.79 KB)		
Non-trainable params: 0 (0.00 Byte)		

سپس مدل را کامپایل می‌کنیم. در این مرحله با استفاده از `categorical_crossentropy` بهینه‌سازی را تعیین کنیم. از آنجا که مسئله از نوع `multi-class classification` است از `adam` به عنوان `optimizer` استفاده شده است که تقاضت میان احتمال کلاس پیش‌بینی شده توسط مدل و کلاس درست (لیبل) آن را محاسبه می‌کند و سعی می‌کند این مقدار را `minimize` کند. همچنین از `fast` به عنوان `efficiency` بهره‌مندی دارد، سریع همگرا می‌شود (convergence).

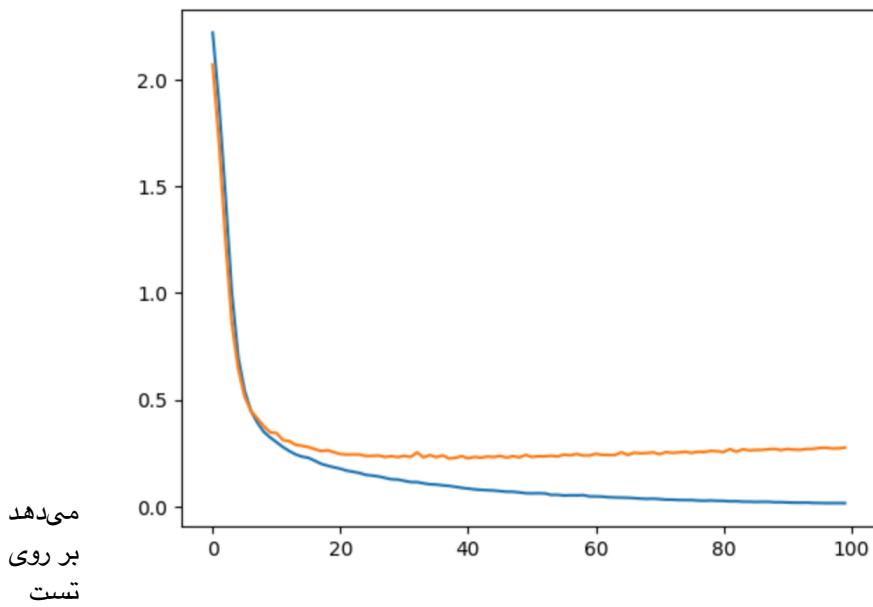
```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

در نهایت مدل را بر روی داده‌ها آموزش داده و تست می‌کنیم که همانطور که از خروجی پیداست میزان loss در epoch‌های بعدی کاهش و میزان accuracy افزایش یافته است:

```
Epoch 90/100
16/16 [=====] - 0s 4ms/step - loss: 0.0199 - accuracy: 0.9980 - val_loss: 0.2641 - val_accuracy: 0.9400
Epoch 91/100
16/16 [=====] - 0s 5ms/step - loss: 0.0186 - accuracy: 0.9980 - val_loss: 0.2694 - val_accuracy: 0.9450
Epoch 92/100
16/16 [=====] - 0s 4ms/step - loss: 0.0176 - accuracy: 0.9980 - val_loss: 0.2661 - val_accuracy: 0.9400
Epoch 93/100
16/16 [=====] - 0s 5ms/step - loss: 0.0174 - accuracy: 0.9980 - val_loss: 0.2650 - val_accuracy: 0.9450
Epoch 94/100
16/16 [=====] - 0s 5ms/step - loss: 0.0181 - accuracy: 0.9980 - val_loss: 0.2696 - val_accuracy: 0.9350
Epoch 95/100
16/16 [=====] - 0s 5ms/step - loss: 0.0162 - accuracy: 0.9990 - val_loss: 0.2696 - val_accuracy: 0.9400
Epoch 96/100
16/16 [=====] - 0s 5ms/step - loss: 0.0160 - accuracy: 0.9980 - val_loss: 0.2740 - val_accuracy: 0.9400
Epoch 97/100
16/16 [=====] - 0s 7ms/step - loss: 0.0153 - accuracy: 0.9990 - val_loss: 0.2753 - val_accuracy: 0.9400
Epoch 98/100
16/16 [=====] - 0s 7ms/step - loss: 0.0154 - accuracy: 0.9990 - val_loss: 0.2717 - val_accuracy: 0.9350
Epoch 99/100
16/16 [=====] - 0s 7ms/step - loss: 0.0156 - accuracy: 0.9990 - val_loss: 0.2722 - val_accuracy: 0.9400
Epoch 100/100
16/16 [=====] - 0s 8ms/step - loss: 0.0152 - accuracy: 1.0000 - val_loss: 0.2756 - val_accuracy: 0.9450
```

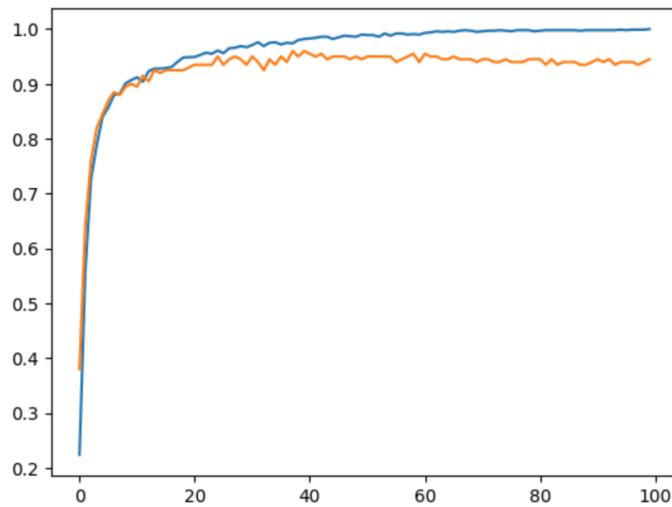
تحلیل نمودارها:

خروجی نمودار تابع ضرر نشان می‌دهد که مقدار ضرر در داده‌های آموزشی همواره روند کاهشی داشته اما در داده‌های تست با آنکه به طور کلی روند کاهشی داشته اما دچار نوسانات بیشتری بوده و در ایپوکه‌های آخر با شیب بسیار کم روند افزایشی را نیز تجربه کرده است. همچنانی همانطور که پیش‌بینی می‌شود مقدار ضرر نهایی ضرر بر روی داده‌های آموزشی کمتر از داده‌های تست است.



نمودار دقت نشان که روند کلی دقت مدل هم داده‌های آموزشی و هم کاهش نیز یافته است.

همچنانی مقدار نهایی دقت بر روی داده‌های آموزشی بیشتر از داده‌های تست است.



۵. فایل Q2_5 ضمیمه شده است. به توضیح کدهای اضافه شده میپردازم:

ابتدا ورودی و برچسبها را با توجه به عملگر منطقی XNOR تعریف میکنیم:

```
input_x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
output_y = np.array([[1], [0], [0], [1]])
```

سپس ساختار MLP را مشخص میکنیم. این مدل ساده دارای دو لایه است و نرخ یادگیری آن 1×10^{-3} در نظر گرفته شده است. وزن‌ها و بایاس‌های اولیه به صورت رندوم انتخاب شده‌اند:

```
# Structure of my MLP
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.1

# Initialization
np.random.seed(0)
w1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
w2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))
```

سپس تابع فعالسازی استفاده شده یعنی Sigmoid را پیاده‌سازی می‌کنیم و به آموزش مدل می‌پردازیم. تعداد epoch‌ها را ۵۰۰۰ در نظر گرفت‌ایم و هر ۱۰۰۰ ایپوکی که مدل جلو می‌رود میزان loss آن را چاپ می‌کنیم.

فرایند آموزش به این صورت است که ورودی‌ها را در وزن‌ها ضرب کرده، با بایاس جمع می‌کنیم. سپس آن را از تابع فعالسازی رد کرده و به لایه بعد می‌دهیم. در مرحله back propagation نیز میزان loss را حساب می‌کنیم و با توجه به گرادیان آن وزن‌ها و بایاس‌ها را آپدیت می‌کنیم.

```
[5] # Train
epochs = 5000

for epoch in range(epochs):
    # Forward pass
    layer1_input = np.dot(input_x, w1) + b1
    layer1_output = sigmoid(layer1_input)
    layer2_input = np.dot(layer1_output, w2) + b2
    layer2_output = sigmoid(layer2_input)

    # Back Propagation
    loss = output_y - layer2_output
    d_layer2 = loss * sigmoid_derivative(layer2_output)
    d_layer1 = d_layer2.dot(w2.T) * sigmoid_derivative(layer1_output)

    # Update
    w2 += layer1_output.T.dot(d_layer2) * learning_rate
    b2 += np.sum(d_layer2, axis=0, keepdims=True) * learning_rate
    w1 += input_x.T.dot(d_layer1) * learning_rate
    b1 += np.sum(d_layer1, axis=0, keepdims=True) * learning_rate

    if epoch % 1000 == 0:
        print(f'Epoch {epoch}, Loss: {np.mean(np.abs(loss))}')

Epoch 0, Loss: 0.496056245511035
Epoch 1000, Loss: 0.3923446900817431
Epoch 2000, Loss: 0.22079588481204324
Epoch 3000, Loss: 0.12337845340033848
Epoch 4000, Loss: 0.08812264266118598
```

در مرحله تست نیز ورودی‌های قبلی را به مدل می‌دهیم و مدل خروجی‌ها را پیش‌بینی می‌کند. همانطور که مشخص است مدل با دقت مناسبی خروجی‌ها را پیش‌بینی کرده است.

```
# Test
layer1_input = np.dot(input_x, w1) + b1
layer1_output = sigmoid(layer1_input)
layer2_input = np.dot(layer1_output, w2) + b2
layer2_output = sigmoid(layer2_input)

print("Predicted Output:")
print(layer2_output)
```

```
Predicted Output:
[[0.95410832]
 [0.07172561]
 [0.07408679]
 [0.90962158]]
```

که ارائه شده نشانگر یک شبکهای عصبی ژرف (Deep Learning) به کمک کتابخانه TensorFlow و مجموعه داده MNIST برای دسته‌بندی تصاویر ارقام از ۰ تا ۹ می‌باشد.

ابتدا package مورد نیاز برای کار با TensorFlow و رسم نمودارها با کتابخانه matplotlib وارد می‌کنیم. سپس از دیتابست MNIST با استفاده از mnist.load_data داده‌های آموزش و آزمون را load می‌کنیم. سپس تصاویر را به مقیاس مقادیر پیکسلها را در بازه‌ی ۰ تا ۱ تبدیل می‌کنیم.

برچسبها را به صورت یک بردار دوویی (one-hot encoded) با کمک to_categorical تبدیل می‌کنیم تا برای هر کلاس یک عنصر برای برچسب متناظر باشد.

یک مدل شبکه عصبی چندلایه (MLP) با استفاده از TensorFlow تعریف می‌کنیم. این مدل دارای لایه‌ای Flatten (برای تبدیل تصاویر به بردار) و چند لایه Dense (لایه‌ای کاملاً متصل) است. لایه‌ای Activation به ترتیب ReLU و softmax برای لایه‌ای مخفی و خروجی اضافه می‌کنیم.

مدل را با تنظیمات نهایی مانند بهینه‌ساز 'adam', تابع خطا 'categorical_crossentropy' و معیارهای متريک 'دقت' مطابق با مسئله دسته‌بندی چند کلاسه، کامپایل می‌کنیم.

مدل را بر روی داده‌ای آموزش آموزش می‌دهیم (train) تا وزن‌ها بهینه شوند. این فرآیند به مدل یاد می‌دهد که تصاویر را به درستی دسته‌بندی کند.

سپس، نمودارهای مربوط به عملکرد مدل را رسم می‌کنیم. این نمودارها شامل نمودار تغییرات خطا در طول زمان آموزش و نمودار تغییرات دقت در طول زمان آموزش و ارزیابی (آزمون) می‌شوند.

در مرحله آخر، عملکرد مدل را بر روی داده‌ای آزمون ارزیابی می‌کنیم و دقت دسته‌بندی را گزارش می‌دهیم. این کد نشان دهنده یک فرآیند ساده از آموزش یک مدل عصبی برای تشخیص ارقام از تصاویر دیجیتال است و می‌تواند به عنوان یک پایه برای پروژه‌های یادگیری عمیق و دسته‌بندی تصاویر مورد استفاده قرار گیرد.

انتخاب تعداد و سایز لایه‌ها بنا به پیچیدگی و شرایط مسئله می‌تواند متفاوت باشد. برای مثال اینجا ورودی $24 \times 24 = 784$ و خروجی ۱۰ تایی است. بنابراین سایز لایه‌ای میانی را بین این دو عدد انتخاب کرده‌ایم:

لایه ورودی: این لایه به طور ضمنی با تعیین شکل ورودی به (28، 28) تعریف می‌شود. به عنوان یک لایه مجزا در Keras ایجاد نمی‌شود. لایه ورودی داده‌ای تصویری به ابعاد 28x28 پیکسل را به یک بردار 1D به اندازه 784 تبدیل می‌کند.

لایه Flatten (تراز کردن): این لایه با استفاده از model.add(Flatten(input_shape=(28, 28))) ایجاد می‌شود. وظیفه این لایه تبدیل داده‌ای تصویری 2D به یک بردار 1D به اندازه 784 است. این لایه معمولاً در کار با داده‌ای تصویری به کار می‌رود تا مقادیر پیکسلها را به یک فرمت مناسب برای لایه‌ای کاملاً متصل تبدیل کند.

لایه Dense (128 نورون): اولین لایه کاملاً متصل با 128 نورون و فعالساز ReLU با استفاده از model.add(Dense(128, activation='relu')) ایجاد می‌شود.

لایه Dense (64 نورون): دومین لایه کاملاً متصل با 64 نورون و فعالساز ReLU با استفاده از model.add(Dense(64, activation='relu')) ایجاد می‌شود.

لایه Dense (10 نورون): لایه نهایی کاملاً متصل لایه خروجی است که دارای 10 نورون می‌باشد. هر نورون به یکی از اعداد (0 تا 9) متناظر با یکی از ارقام دیجیتال اختصاص داده شده است. این لایه از تابع فعالساز softmax برای دسته‌بندی چند کلاسه استفاده می‌کند: (.model.add(Dense(10, activation='softmax')))

بنابراین، در این معماری:

لایه ورودی به صورت صریح در تعداد لایه‌ها در نظر گرفته نمی‌شود، اما در تبدیل داده‌ها به منظور پردازش بیشتر نقش دارد.

دو لایه مخفی با 128 و 64 نورون به ترتیب وجود دارد که برای یادگیری ویژگی‌های سلسله مراتبی از داده‌های ورودی استفاده می‌شوند.

لایه خروجی 10 نورون دارد که برای دسته‌بندی تصویر ورودی به یکی از 10 کلاس ممکن (0 تا 9) استفاده می‌شوند. انتخاب تعداد لایه‌ها و اندازه نورون‌ها در هر لایه می‌تواند بر اساس مسئله خاص، داده‌ها و منابع محاسباتی موجود متغیر باشد. در این مثال، این یک معماری نسبتاً ساده برای مجموعه داده MNIST است، اما در عمل ممکن است نیاز باشد با معماری‌ها و اندازه‌های مختلف برای بهینه‌سازی عملکرد در وظیفه خاص خود آزمایش کنیم.

نمودار خروجی: همانطور که مشخص است دقت مدل طی آموزش افزایش یافته و نهایتاً به ۹۷.۶۷٪ رسیده است.

