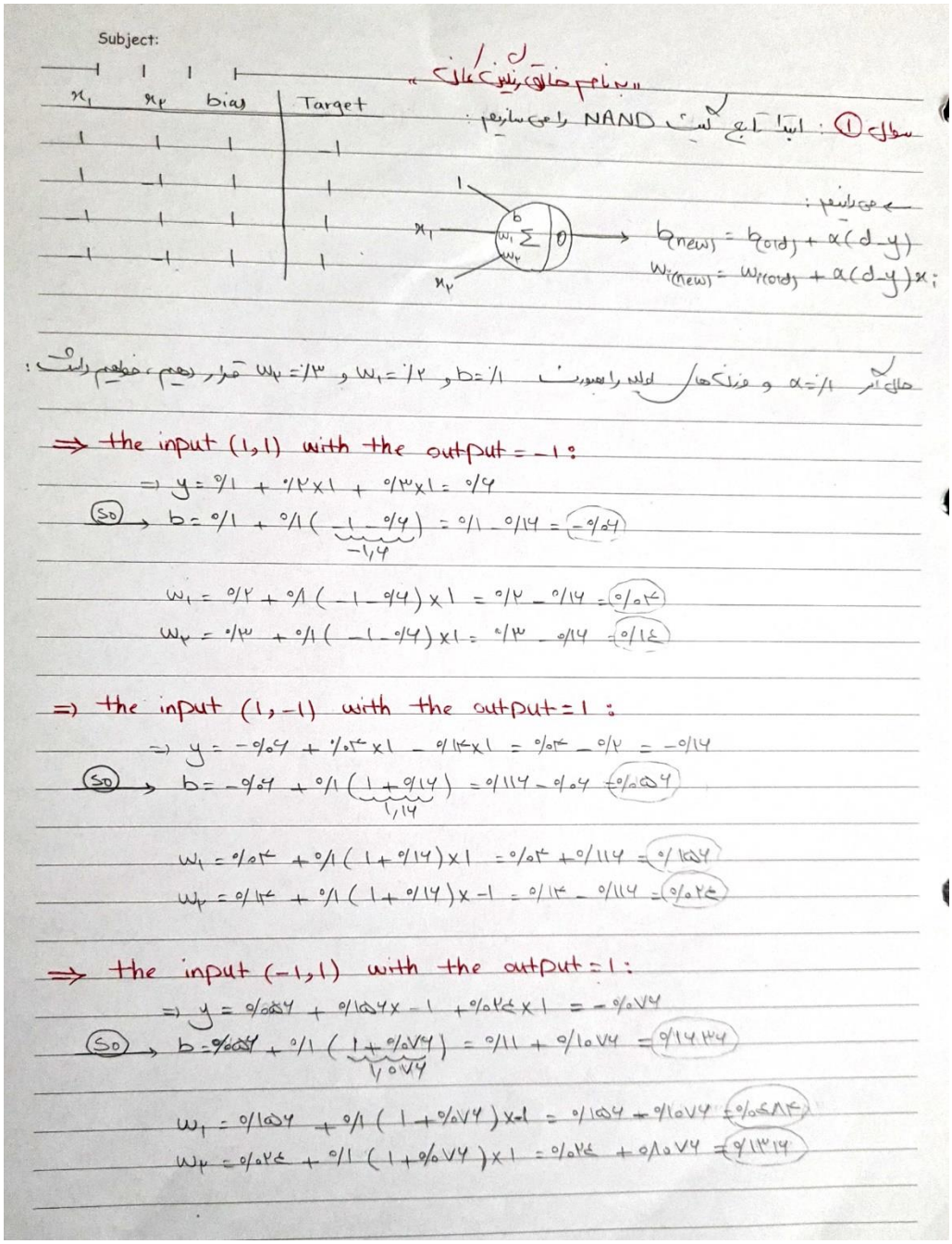
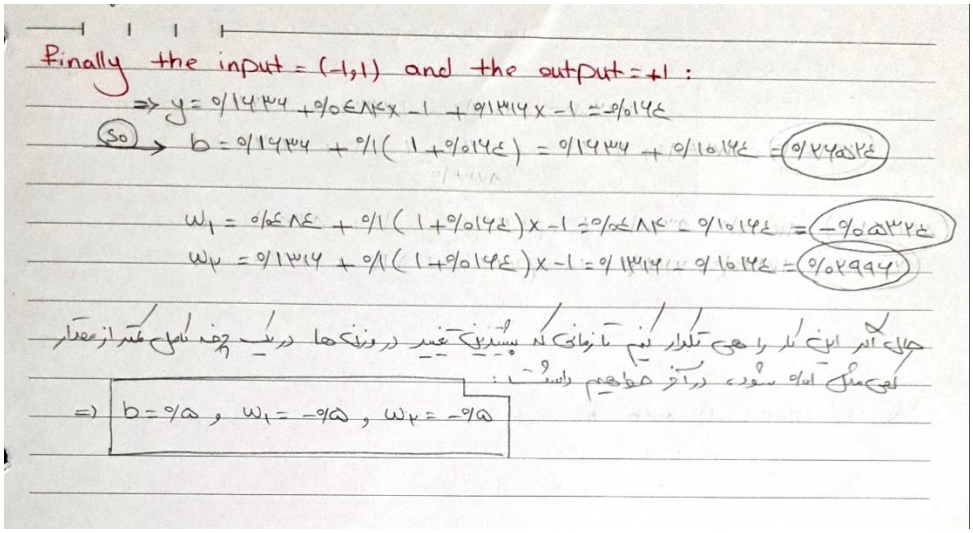
به نام خالق رنگین کمان

ستاره باباجانی99521109- گزارش تمرین دوم





سوال2: الف)

- تابع فعال‌سازی خطی: یک تابع فعال‌سازی خطی توسط یک تابع خطی ساده نمایان می‌شود. یعنی خروجی این تابع خطی از جمع وزن‌های ورودی‌ها به نسبت ورودی‌ها است.(شبکه یک رابطه خطی بین ویژگی های ورودی و متغیر هدف می آموزد.) معادله تابع فعال‌سازی خطی به شکل زیر است:

$$f(x) = a * x + b$$

این توابع مناسب برای وظایف ساده‌تری مانند رگرسیون هستند (که مقدار پیوسته پیش بینی میشود) زیرا محدود به توانایی مدل‌کردن الگوهای پیچیده‌تر در داده‌ها هستند.

- تابع فعال‌سازی غیرخطی: توابع فعال‌سازی غیرخطی اغلب به عنوان توابع غیرخطی شناخته می‌شوند، زیرا خروجی آن‌ها ترکیبی غیرخطی از ورودی‌ها و وزن‌ها است. توابع فعال‌سازی غیرخطی مانند تابع سیگموید، تانژانت هیپربولیک، و تابع ReLU (ویرایش شده یا غیر ویرایش شده) هستند.

  توابع غیرخطی به شبکه‌ها این امکان را می‌دهند که الگوهای پیچیده‌تری را در داده‌ها مدل کنند و ویژگی‌های غیرخطی را استخراج کنند.(پس در شبکه های عصبی که روابط پیچیده ای بین ورودی و خروجی وجود دارد، مهم هستند) این می‌تواند به عملکرد بهتر در وظایف تصویربرداری، تشخیص الگو، و ترجمه ماشینی کمک کند.

انتخاب تابع فعال‌سازی، معمولاً وابسته به وظیفه مورد نظر و نوع داده‌ها است. توابع فعال‌سازی غیرخطی برای وظایف پیچیده و متنوع معمولاً انتخاب مناسب‌تری هستند.

ب)

1. بایاس رندوم و وزن‌ها صفر:

- وزن‌های صفر به معنی این است که همه ورودی‌ها به هیچ‌کدام از نورون‌ها تأثیری ندارند. این به معنی این است که شبکه هیچ‌کاربردی نخواهد داشت.

- بایاس‌های رندوم به معنی این است که نورون‌ها در لایه‌ها به یک اندازه تأثیر دارند، اما تأثیر تمامی ورودی‌ها صفر است. به عبارت دیگر، همه نورون‌ها یک خروجی یکسان دارند.

- در این حالت، آموزش به سرعت متوقف می‌شود چرا که شبکه نمی‌تواند الگوهای مهم در داده‌ها را یاد بگیرد. باید وزن‌ها به صورت تصادفی ایجاد شوند تا شبکه بتواند در فرآیند یادگیری تغییرات مناسبی ایجاد کند.

2. بایاس صفر و وزن‌ها رندوم:

- در این حالت، همه نورون‌ها به یک اندازه تأثیر دارند(قدرت تعمیم به داده های تست محدود میشود)، و ورودی‌ها تأثیر متنوعی برای نورون‌ها ایجاد می‌کنند.

- آموزش از این حالت شروع می‌شود و وزن‌ها به مرور زمان بهبود می‌یابند. با این حال، وزن‌های تصادفی می‌توانند منجر به آموزش کند، کارهای طولانی مدت و اشکالات آموزشی شوند.

- برخی از وزن‌ها ممکن است به ترتیب خاصی ایجاد شوند که باعث ایجاد مشکلاتی در شبکه شود. به عنوان مثال، اگر تمام وزن‌ها به یک مقدار تصادفی مشابه تنظیم شوند، می‌تواند دچار مشکل شود.

به طور کلی، بهتر است وزن‌ها و بایاس‌ها به صورت تصادفی با مقادیر کوچک و متنوع مقداردهی اولیه شوند تا آموزش شبکه به صورت موثر آغاز شود و فرآیند بهبود وزن‌ها از طریق الگوریتم‌های بهینه‌سازی مانند نزول گرادیان، به جهت بهبود عملکرد شبکه انجام شود.

ج) منظور از توانایی تعمیم، عملکرد خوب مدل روی داده های جدیدی است که قبلا ندیده است.

1. MLP (Multi-Layer Perceptron):

- MLP یک مدل عصبی چند لایه است و به عنوان یک مدل قوی و کارا در بسیاری از وظایف معمولی مورد استفاده قرار می‌گیرد.
- MLP با تنظیمات مناسب و انتخاب معماری مناسب می‌تواند قابلیت تعمیم خوبی داشته باشد.(الگو های پیچیده را در داده ها ثبت کرده و آنها را تعمیم میدهد.)

2. Kohonen (SOM - Self-Organizing Map):

- SOM یک نوع شبکه عصبی خودسازمانده است و عمدتاً برای مسائل دسته‌بندی و نگاشت تصویری مورد استفاده قرار می‌گیرد.
- SOM به خوبی می‌تواند الگوهای مکرر در داده‌ها را تشخیص دهد و تصویرهای مفهومی ایجاد کند.

3. Madaline (McCulloch-Pitts Neuron):

- Madaline یک نورون مک‌کالاک پیتس ساده است و معمولاً برای وظایف ساده مورد استفاده قرار می‌گیرد.
- Madaline به صورت پایه‌ای عمل می‌کند و توانایی تعمیم به وظایف پیچیده‌تر را ندارد.

4. Perceptron:

- Perceptron نیز یک نورون ساده و برای دسته‌بندی ساده‌تری مورد استفاده قرار می‌گیرد.
- از نظر تعمیم‌پذیری، مشابه به Madaline عمل می‌کند و برای وظایف پیچیده‌تر به شبکه‌های عمیق‌تر نیاز است.

5. Adaline (Adaptive Linear Neuron):

- Adaline نیز یک نورون خطی تطبیقی است و برای وظایف مشابه به Perceptron و Madaline مورد استفاده قرار می‌گیرد.

پس به صورت کلی اگر شبکه های عصبی را طبق توانایی تعمیمشان رتبه بندی کنیم، خواهیم داشت:

1) MLP
2) Kohonen
3) Madaline, Adaline, Perceptron

د)استفاده از معکوس ماتریس هسین برای تعیین تغییر وزن در هر مرحله از آموزش شبکه MLP یک روش معتبر و موثر در یادگیری ماشینی است. این روش اغلب به عنوان یک روش بهینه‌سازی دومین مرتبه یا بهینه‌سازی Newton-Raphson شناخته می‌شود.

مزایا:

1. سرعت همگرایی: از مزیت‌های بزرگ استفاده از معکوس ماتریس هسین در بهینه‌سازی تانژانت یا تعیین تغییرات وزن، سرعت همگرایی بسیار سریع تر نسبت به روش‌های گرادیان نزولی معمولی است که در مسائلی که نیاز به تعداد دقیق تغییرات کم دارند مورد توجه قرار می‌گیرد.(علت این سرعت، استفاده از انحنای سطح خطا است.)

2. دقت بالا: با استفاده از اطلاعات دوم مشتق (درجه دوم) از تابع خطا نسبت به وزن‌ها، میتوان به دقت بالاتری در بهینه‌سازی شبکه‌ها رسید

یعنی ممکن است به نقاط بهینه‌تری نسبت به روش‌های گرادیان نزولی ساده رسید.

3. استحکام بیشتر: در حالاتی که سطح خطا بسیار خطی است، روش های مرتبه دوم میتوانند حتی با سطوح خطای نامنظم به همگرایی منجر شوند.

معایب:

1. پیچیدگی محاسباتی: محاسبه معکوس ماتریس هسین می‌تواند بسیار پرمصرف باشد، به ویژه در شبکه‌های عصبی با تعداد وزن‌های بسیار زیاد.

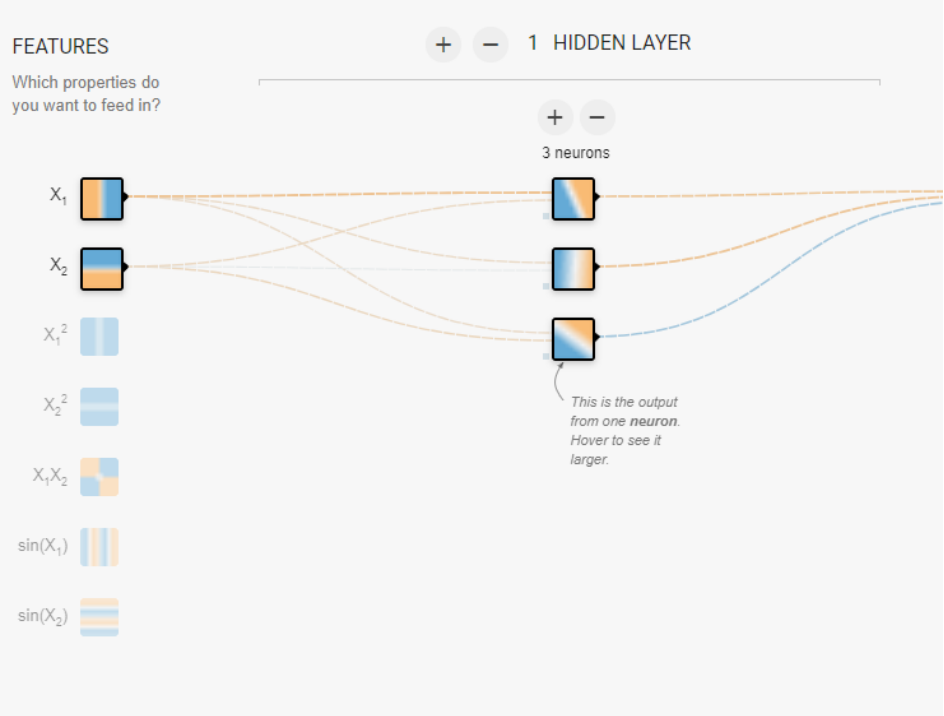
2. ذخیره‌سازی ماتریس هسین(نیاز به حافظه زیاد): ذخیره‌سازی و مدیریت ماتریس هسین برای شبکه‌های بزرگ میتواند به حافظه بسیار زیادی نیاز داشته باشد.

3. آموزش بدون نقطه توقف: معکوس ماتریس هسین ممکن است در مواردی که هسین نامعتبر است یا به توقف نمی‌رسد (به علت دورزنی و یا بی‌بهره بودن ماتریس هسین) به مشکل منجر شود.

4. حساسیت به نویز: این روش معمولاً حساس به نویز در داده‌ها و خطاهای کوچک در محاسبات است که منجر به تخمین نادرست از انحنای سطح خطا میشود.

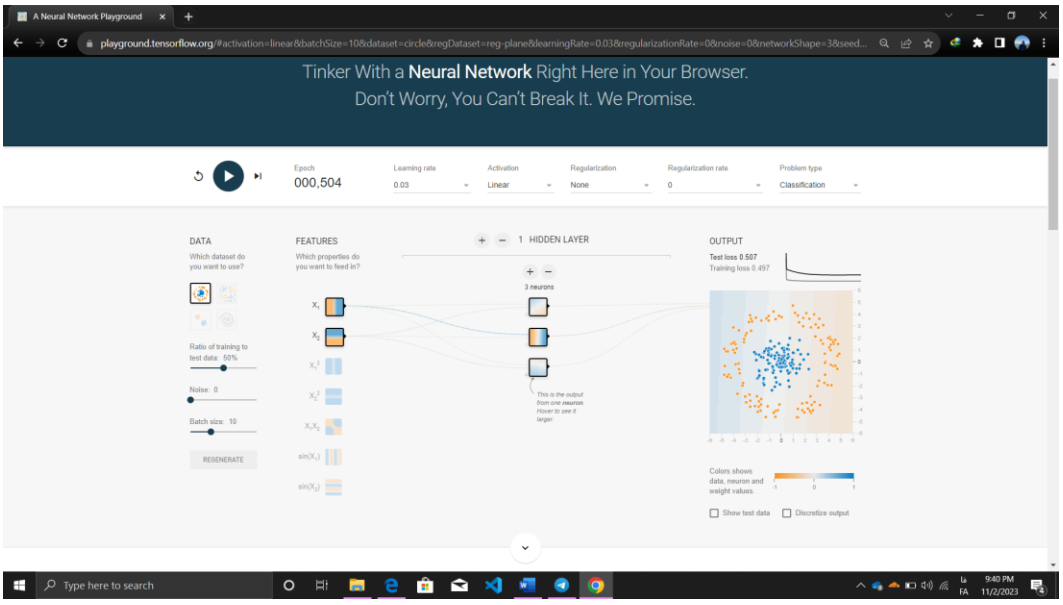
به طور کلی، استفاده از معکوس ماتریس هسین به عنوان یک روش بهینه‌سازی دومین مرتبه در موارد خاص ممکن است مزایا داشته باشد، اما نیاز به مدیریت دقیق و محاسبات پیچیده دارد. بسته به ویژگی‌های مسئله، ممکن است سایر روش‌های بهینه‌سازی مثل نزول گرادیان یا نزول گرادیان نشانه‌ای نیز بهتر عمل کنند.

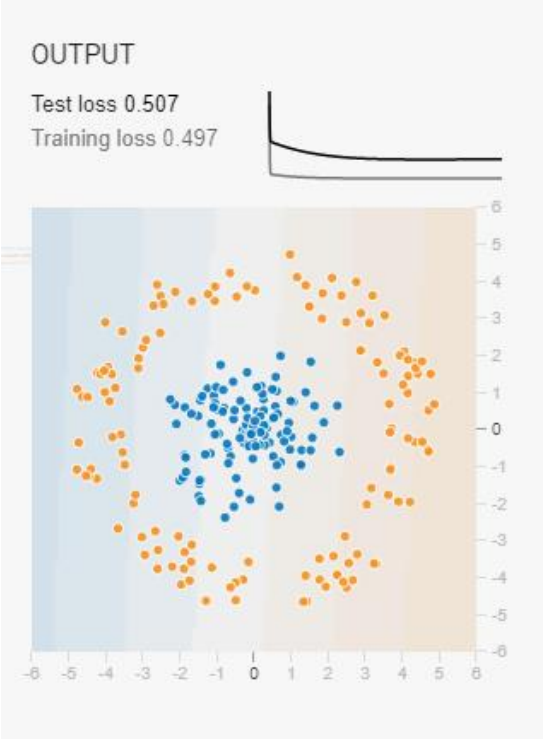
سوال3: در این سوال ابتدا یک لایه پنهان با 3 نورون میانی با فیچر های ورودی x1, x2 ایجاد کردیم. نرخ یادگیری بصورت دیفالت 0.03 است و نوع مشکل، طبقه بندی است. همچنین اندازه هر batch، 10 و مقدار آموزش epoch 500 است. در ادامه تک تک دیتاست ها و تابع های فعال سازی را امتحان کرده و نتیجه را تحلیل میکنیم.



1. تابع linear:

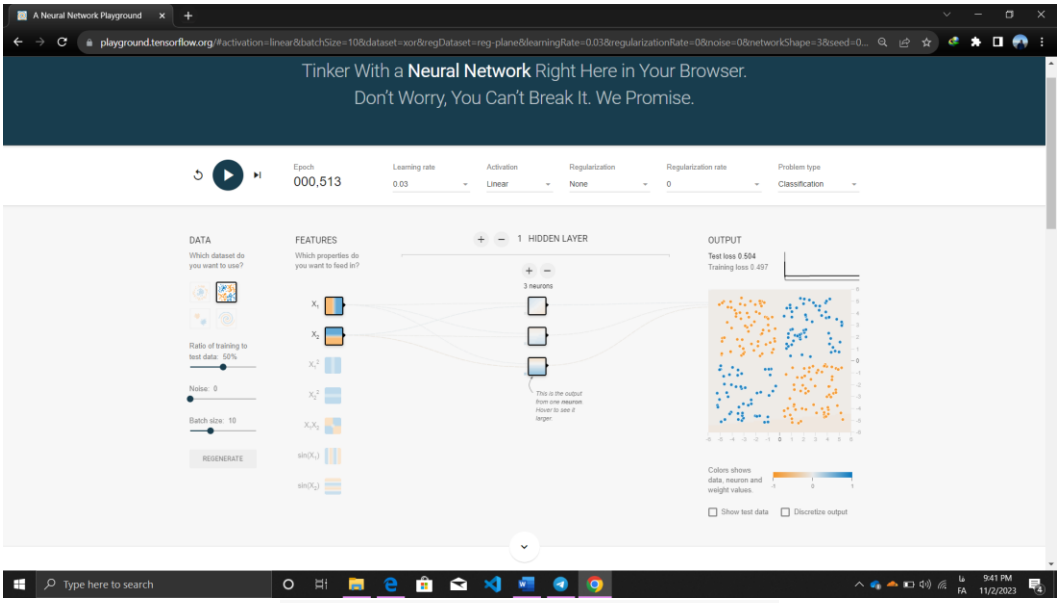
- دیتاست Circle: عکس نتیجه نهایی به شرح زیر است:

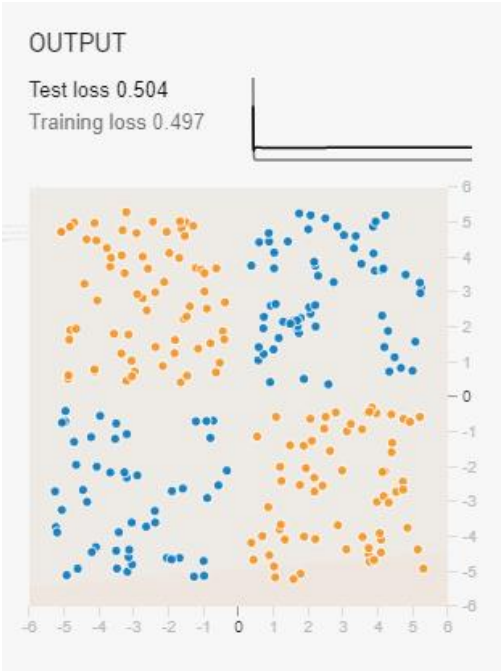




مقدار ضرر چه در داده آموزش و چه در داده تست در گذر زمان تغییری نکرده است و مقدار نسبتا زیادی است به این دلیل که تابع خطی، برای مسائل طبقه بندی استفاده نمیشود چون فقط میتواند الگوهای خطی را نمایش دهد و در این دیتاست نمیتواند انحنای دایره ای شکل ایجاد کند و بیشتر در مشکلات رگرسیون استفاده میشود.

- دیتاست Exclusive: عکس نهایی به شرح زیر است:

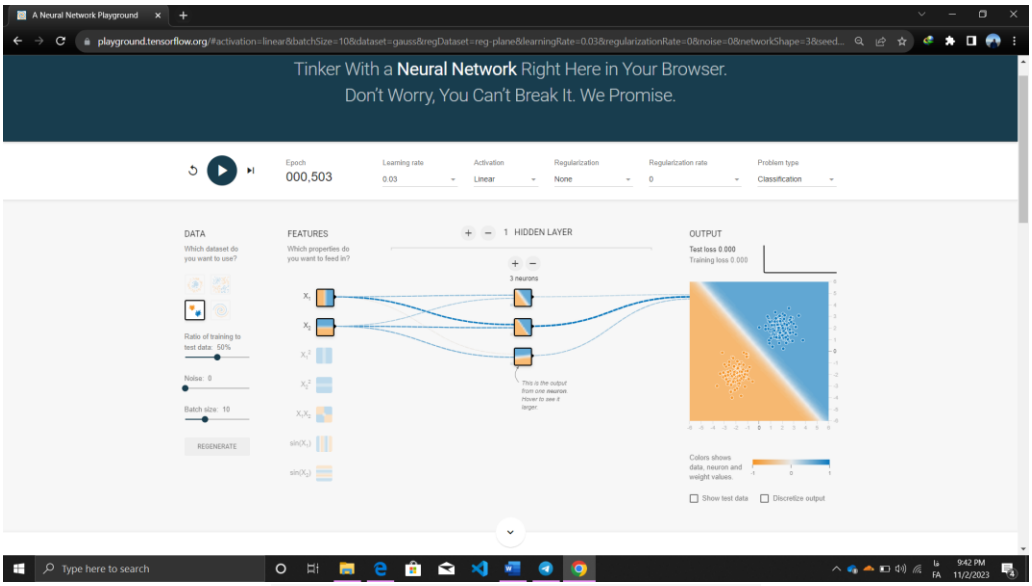




مثل دیتاست قبلی، مقدار ضرر در دو مرحله آموزش و تست زیاد است و در گذر زمان تغییری نکرده است. باز هم علت، عدم توانایی تابع فعال ساز خطی در تولید خروجی غیر خطی است.

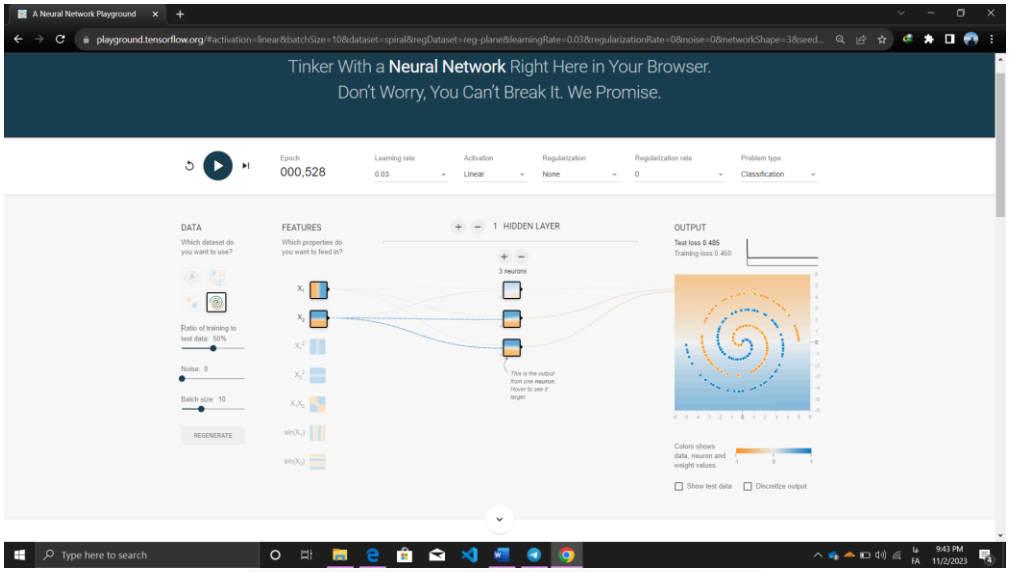
- دیتاست Gaussian: همان طور که در عکس زیر نمایان است، بر روی این دیتاست عملکرد تابع بسیار خوب بوده است زیرا به

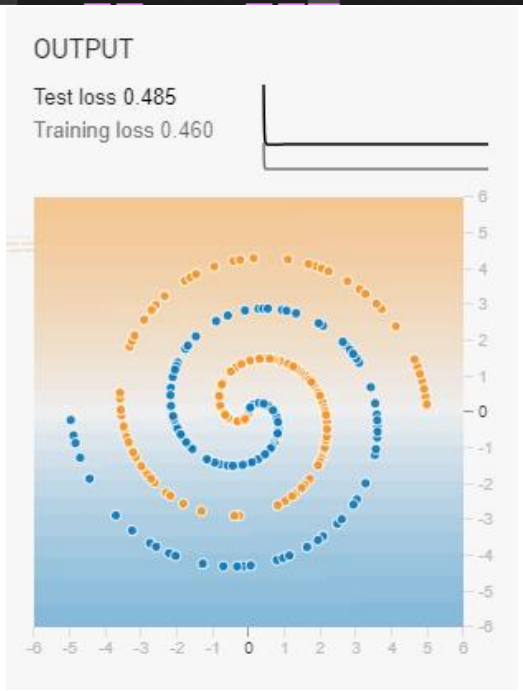
راحتی توانسته یک خط(مرز) بین دو کلاس ایجاد کند.(این اتفاق و صفر شدن مقدار ضرر روی داده آموزش در کمتر از 80 epoch اتفاق افتاد! چون یک دیتاست خطی است.)



- دیتاست Spiral: در این دیتاست هم به دلیل پیچیدگی زیاد مسئله و عدم توانایی تابع خطی در جدا کردن مرز با انحنا،

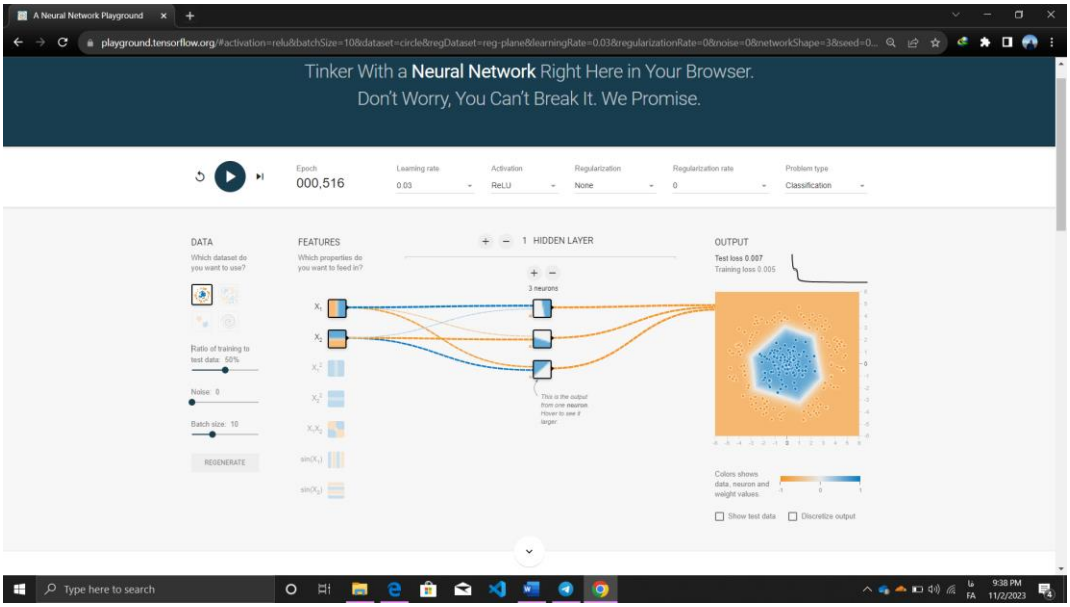
عملکرد تابع ضعیف بوده و مقدار ضررها زیاد بود است و در طول آموزش و گذر زمان، این مقدار تغییری نمیکرد چون تابع توانایی انجام هیچکاری را نداشت.

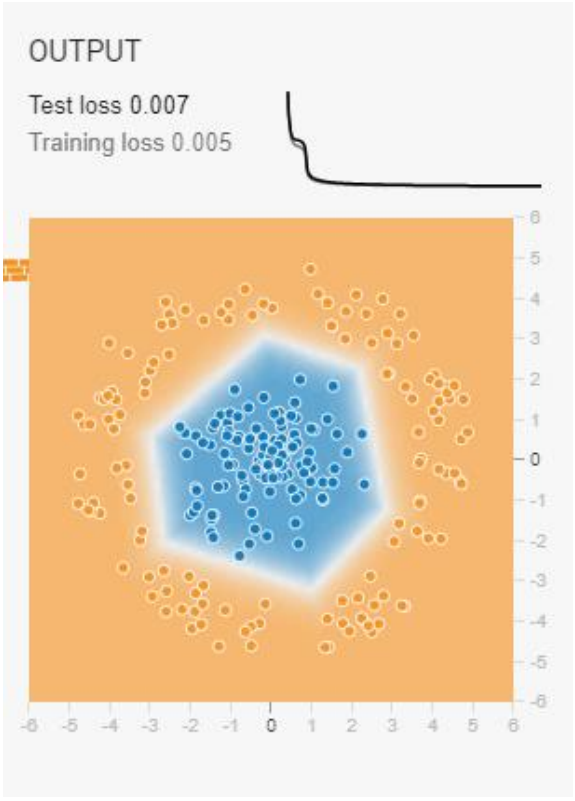




2. تابع RelU:

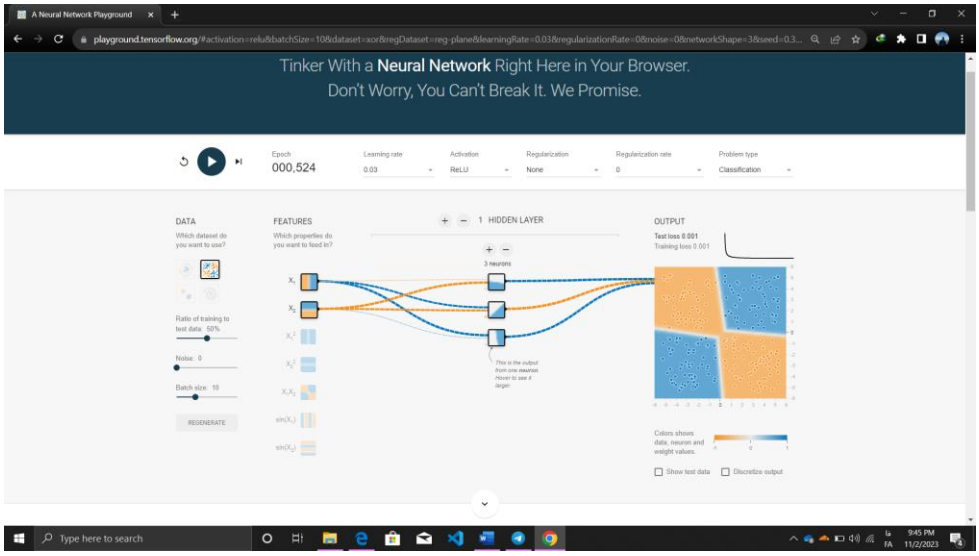
- دیتاست Circle:

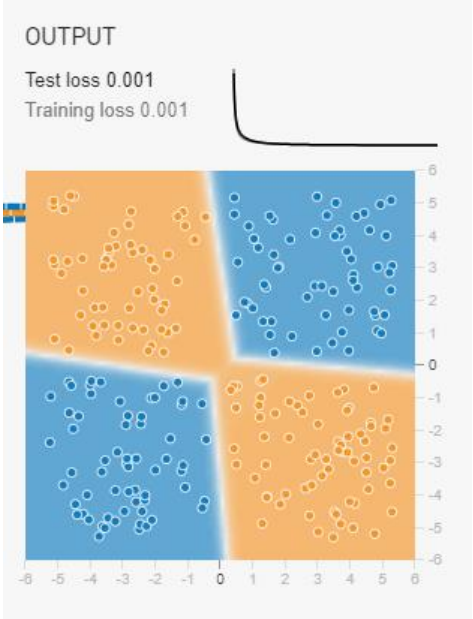




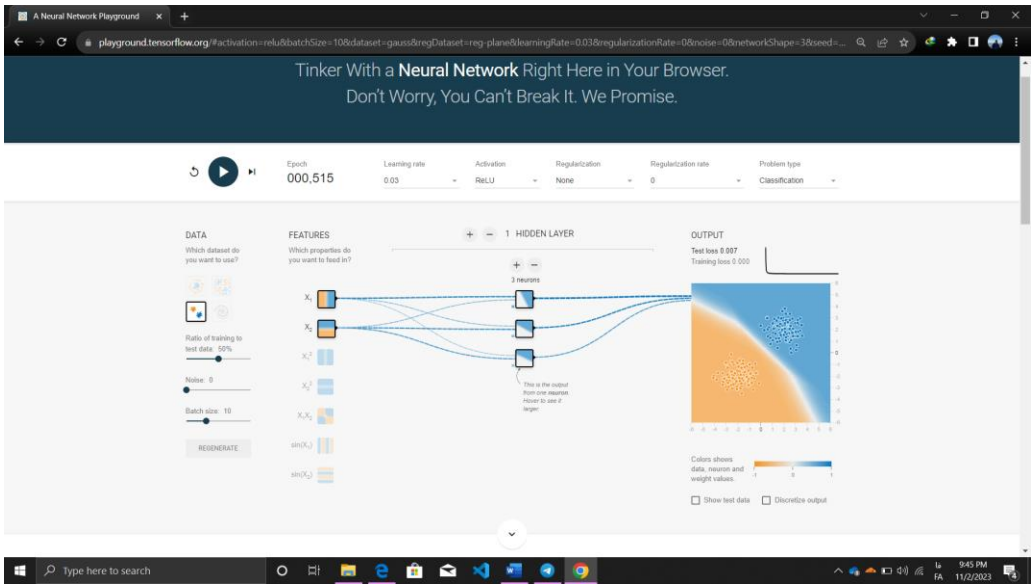
همان طور که از عکس مشاهده میکنید عملکرد تابع بسیار خوب بوده و مقدار ضرر بر روی داده تست و آموزش نزدیک به صفر بوده است.

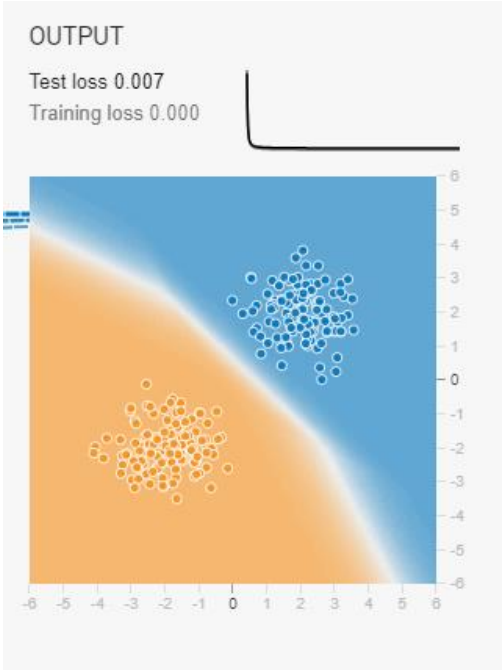
- دیتاستExclusive: همان طور که مشاهده میشود مدل توانسته ارتباط بین کلاس ها را خوب تشخیص دهد آن ها را به خوبی از هم جدا کند. این تابع فعال سازی برای بسیاری از مسائل غیر خطی مناسب است و میتواند جدا کننده مناسبی بین کلاس ها باشد.



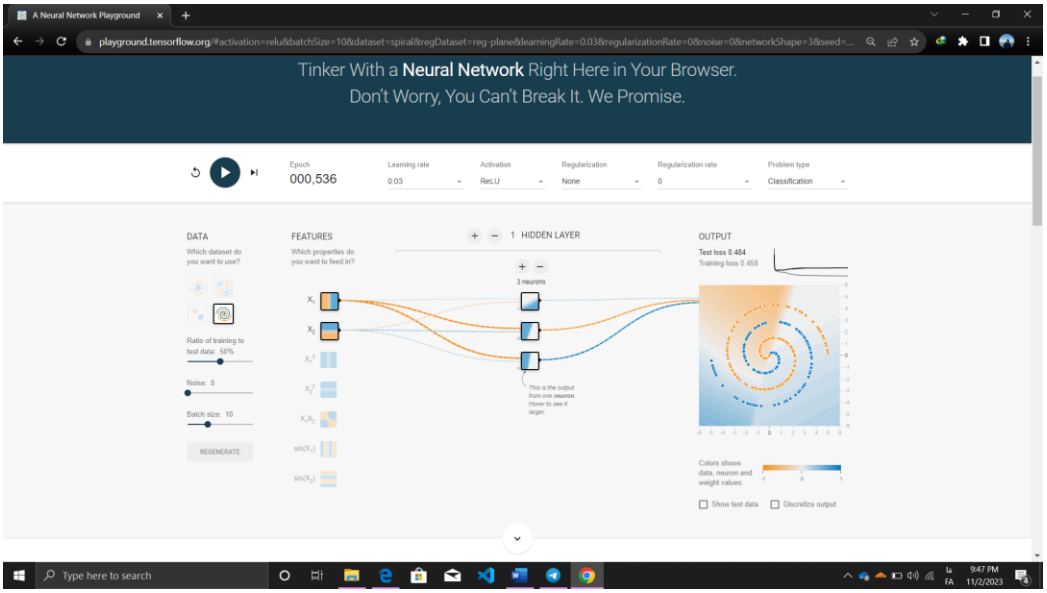


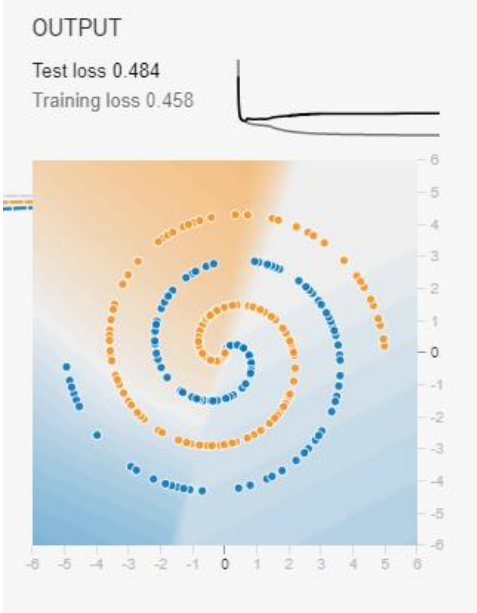
- دیتاستGaussian:برخلاف تابع خطی، در یادگیری با استفاده از این تابع، تعداد epoch های بیشتری صرف شد تا مقدار ضرر به صفر میل کند. ولی همچنان عملکرد این تابع بسیار خوب بوده است و مدل توانسته مرز دقیقی پیدا کند. مقدار تفاوت بین ضرر های داده آموزشی و تست میتواند بخاطر عدم توازن آنها باشد.





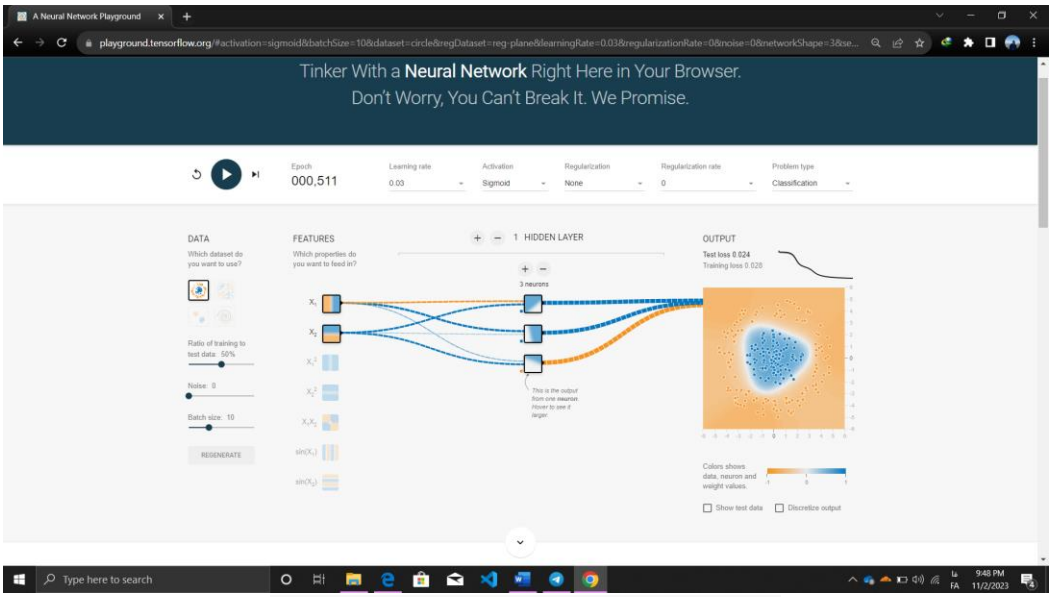
- دیتاستSpiral: همان طور که مشاهده میشود، عملکرد مدل خوب نبوده و نتوانسته مرز را مشخص کند و مقدار ضرر ها زیاد بوده است.(این تابع فعال سازی برای همه ی مسائل غیر خطی مناسب نیست و نمیتواند انقدر انحنا را برای مرز یاد بگیرد!)

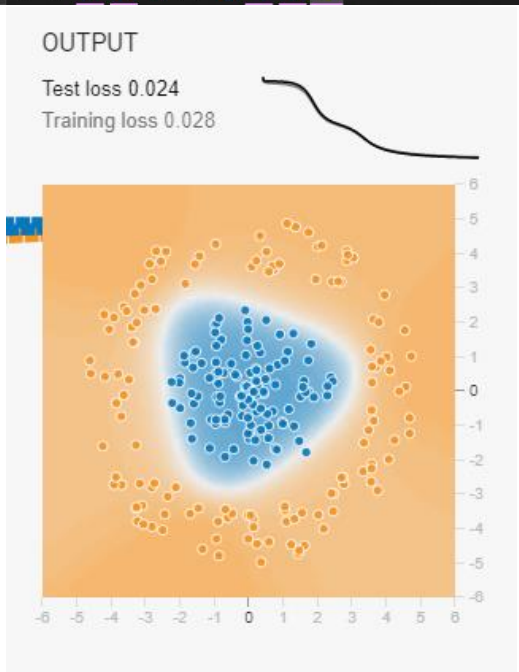




3. تابع Sigmoid:

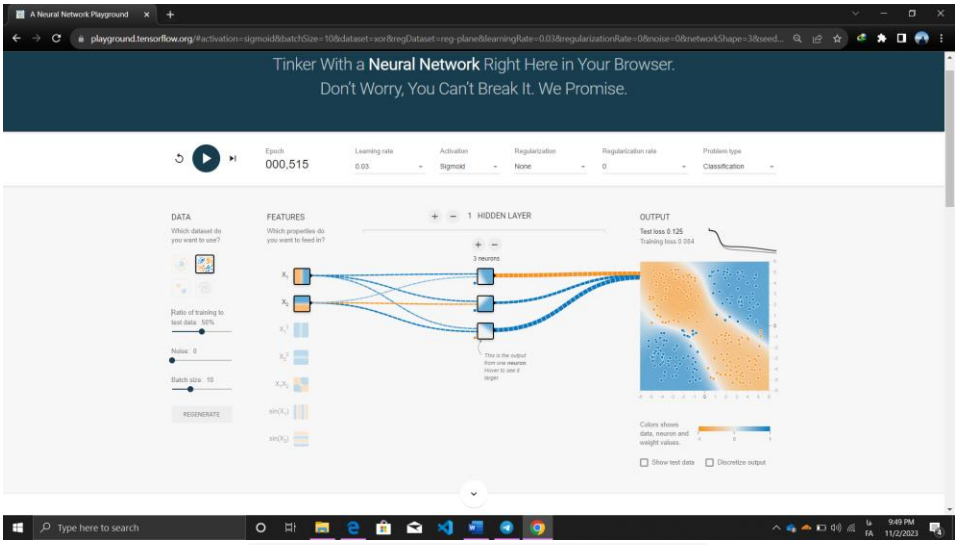
- دیتاستCircle: همان طور که مشاهده میشود در طول epoch ها مقدار ضرر به سرعت کاهش می یافت و در نهایت به مقدار نسبتا کمی رسید زیرا این تابع توانست مرزی مشخص و درست بین دو دایره پیدا کند. هرچند مقدار ضرر آن در مقایسه با RelU بیشتر است.

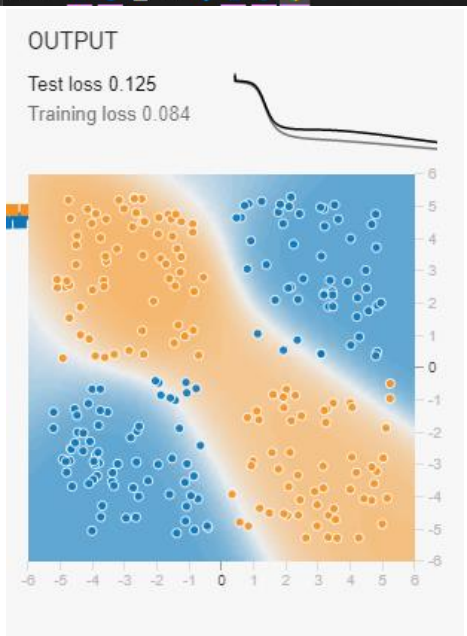




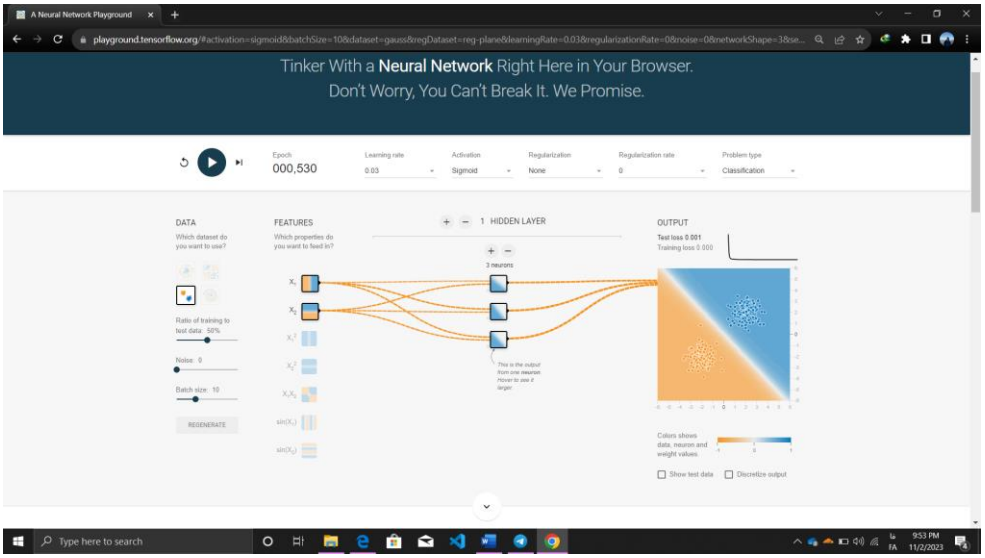
- دیتاستExclusive: همان طور که مشاهده میشود عملکرد تابع روی این دیتاست در داده آموزشی هم خوب بوده است و توانسته در گذر زمان مرز و انحنای خوبی بین کلاس ها پیدا کند.(طبق فرمول سیگموید، یادگیری مرزی که بصورت منحنی باشد راحت است!) مقدار ضرر روی داده تست خیلی بیشتر از داده آموزش

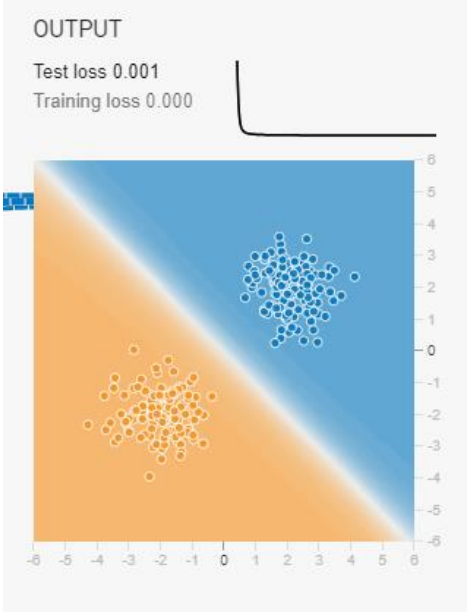
است که ممکن است به علت عدم توازن و درستی پخش داده ها روی آموزش و تست باشد.(مشکل overfitting وجود دارد!)
مقدار ضررها با استفاده از تابع RelU بسیار کمتر بود که به علت این است که مرز در آنجا بصورت خطی و پاره خط بود ولی اینجا منحنی است که باعث میشود یکسری از داده ها درست طبقه بندی نشوند.





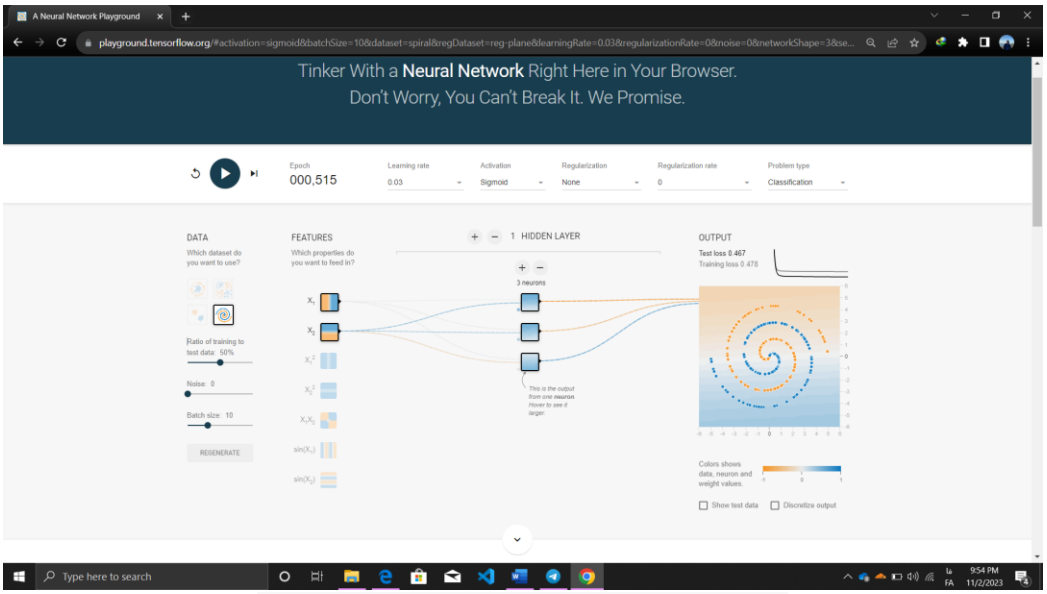
- دیتاستGaussian: مانند تابع RelU، عملکرد سیگموید روی این دیتاست بسیار خوب بوده و ضرری نزدیک به صفر برای داده تست دارد. این دیتاست، دیتاست ساده ای است که برای اکثر توابع فعال سازی، مشخص کردن مرز در آن آسان است. علت اینکه برخلاف تابع خطی، ضرر تست با استفاده از این دو تابع صفر نمیشود، میتواند بخاطر نویز یا اشتباه ریزی که در تعیین مرز وجود دارد باشد که قطعا اگر تعداد epoch ها بیشتر شود، این مشکل رفع میشود.

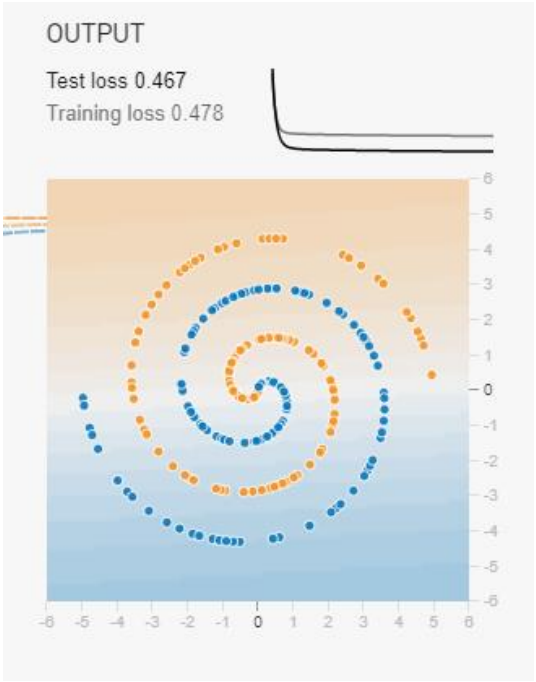




- دیتاستSpiral: همان طور که مشاهده میشود، مرز اصلا درستی انتخاب نشده است که علت آن میتواند بخاطر عدم توانایی این تابع، در درست کردن مرز در اثر برخورد چندین انحنا باشد.(این مسئله، مسئله پیچیده ی غیر خطی است!)

  عملکرد روی داده تست کمی بهتر از داده آموزش بوده است و علت آن میتواند بخاطر عدم توازن در پراکندگی دیتا برای این دو مرحله باشد.

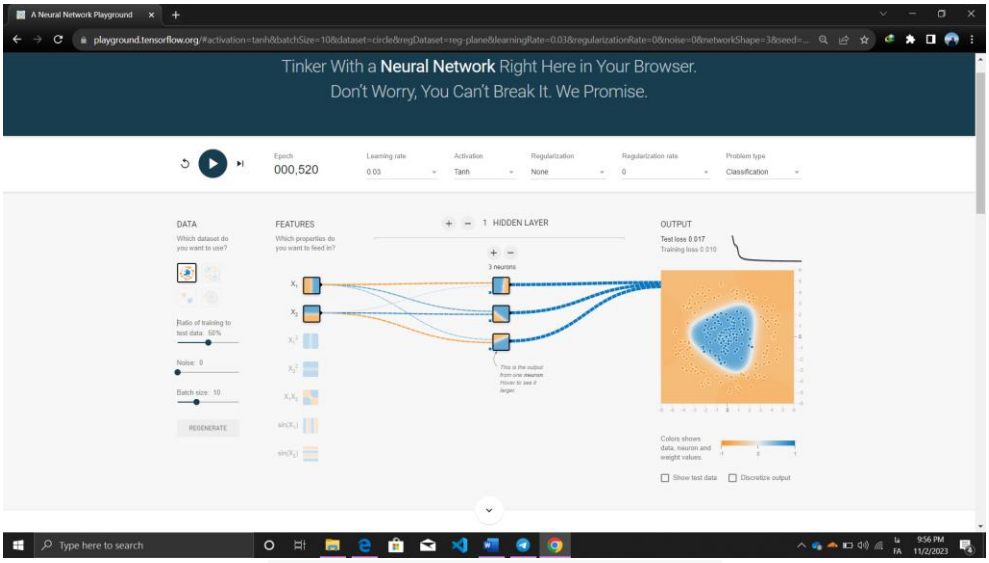


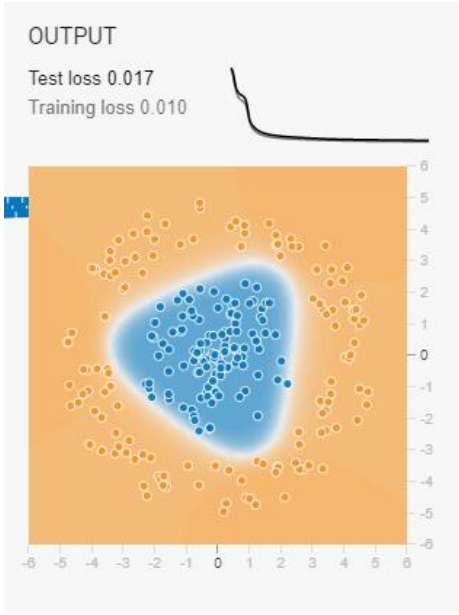


4. تابع Tanh:

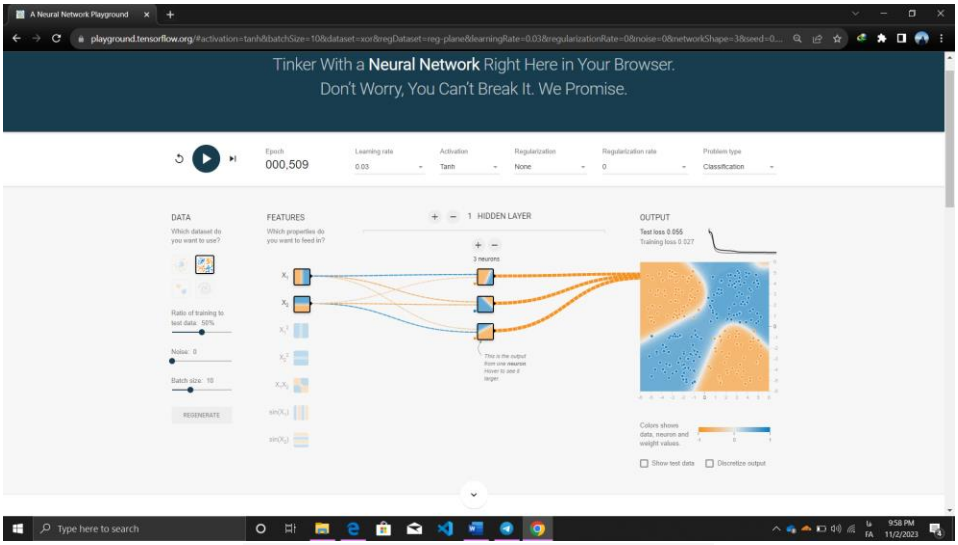
- دیتاستCircle: عملکرد این تابع در این دیتاست بسیار خوب بوده است و همان طور که از شکل مشاهده میشود، مرز بسیار درستی بین دو کلاس تشخیص داده شده است. مقدار ضرر روی داده

تست نزدیک به 0.007 بیشتر است که به نظر مقدار زیادی نیست و با تعداد epoch بیشتر مشکل حل میشود.



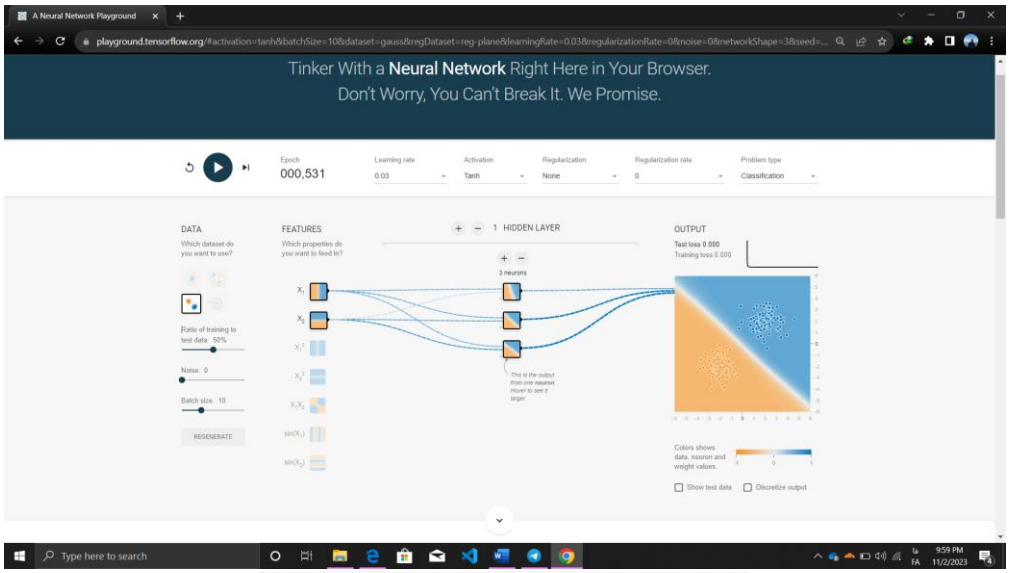


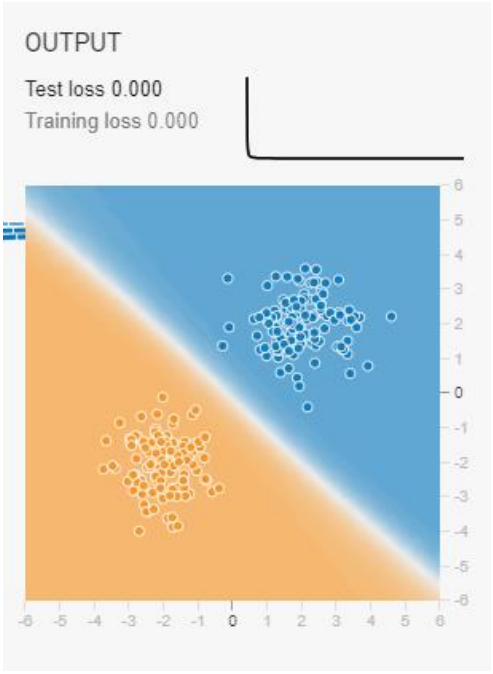
- دیتاستExclusive: تفاوت بسیار زیادی بین مقدار ضرر داده تست و آموزش وجود دارد که میتواند به علت یادگیری بیش از حد روی داده اموزش باشد.



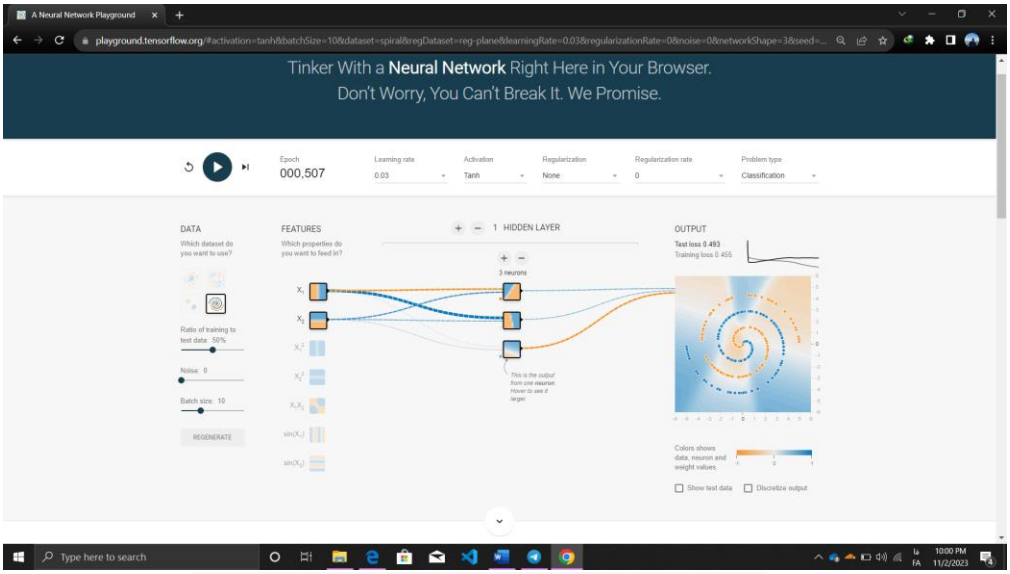
همچنان بهترین عملکرد برای RelU بوده است چون این دو تابع آخر، با مرز منحنی کمی دچار خطا میشدند و دیتاها خوب طبقه نشدند.

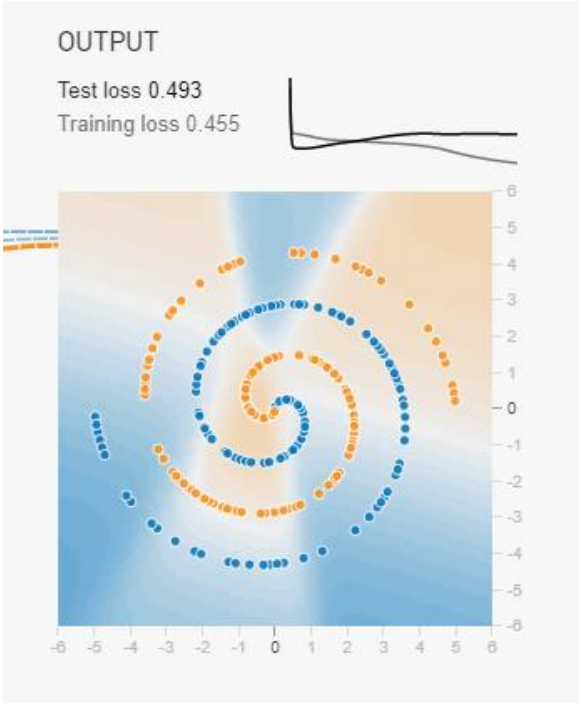
- دیتاستGaussian: مدل با سرعتی بیشتر از استفاده از تابع خطی، مرز را مشخص کرد و به راحتی مقدار ضرر در آموزش و تست را به صفر رساند.(مسئله بسیار ساده است!)





- دیتاستSpiral: عملکرد این مدل نسبت به استفاده از بقیه تابع ها بدتر بوده است زیرا همان طور که از شکل مشاهده میشود مدل سعی داشته مرز را جدا کند ولی بسیاری از دیتا را اشتباه طبقه بندی کرده است.





در اخر، طبق آزمایشات انجام شده میتوان گفت برای دیتاست های Circle, Exclusive, Gaussian, Spiral به ترتیب از چپ به راست، بهترین تابع ها بر روی داده تست این چنین بوده است:

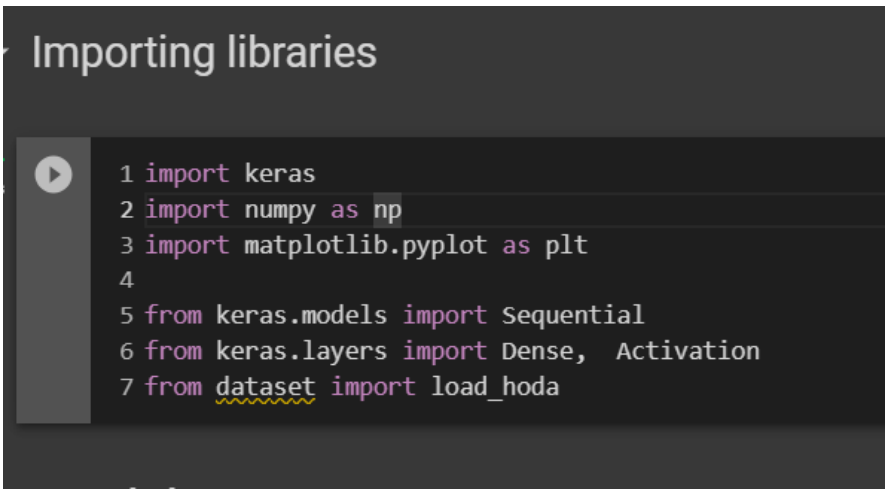
RelU, RelU, Linear and Tanh, Sigmoid

با این حال، تعداد 500 اپاک بسیار برای یادگیری بعضی از این مسئله های پیچیده کم بوده و نیاز به زمان بیشتر و لایه و نورون بیشتر حتما است!

سوال4: به بررسی تک تک سلول ها و نتیجه نهایی میپردازیم:

- در این بخش ابتدا دیتاست را از گوگل دانلود میکنیم:

- سپس کتابخانه های مورد نیاز را صدا میزنیم:



- سپس مقادیر داده train, test را از دیتا ست میگیریم:

Load dataset

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

- حال داده های آموزش و تست را پیش پردازش میکنیم. ابتدا آنها را به یک ارایه numpy تبدیل کرده و سپس به فرم one-hot در می آوریم:

```
# Preprocess input data for Keras.
x_train = np.array(x_train_original) # Convert the training data to a NumPy array
y_train = keras.utils.to_categorical(y_train_original, num_classes=10) # Convert the training labels to one-hot encoding format
x_test = np.array(x_test_original) # Convert the testing data to a NumPy array
y_test = keras.utils.to_categorical(y_test_original, num_classes=10) # Convert the testing labels to one-hot encoding format
```

- در دو سلول بعدی، فرمت و shape داده های آموزش و تست، قبل و بعد از پیش پردازش نمایش داده شده است:

```python
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]))
```

```python
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:
        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.]
```

همان طور که در خروجی مشخص است، ما 1000 نمونه با 25 فیچر برای آموزش و 200 نمونه به 25 فیچر برای تست داریم. تغییر اساسی بعد پیش پردازش در ابعاد y آموزش و تست بود که بیانگر این است که به ازای هر کلاس یک one-hot داریم که تعداد آن 10 است.

- بعد از پیش پردازش، normalization انجام میشود که در طی آن، تایپ دیتای آموزش و تست Float32 شده و با تقسیم بر 255 آن را نرمالیزه میکنیم:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255 # Normalize the training data by dividing by 255
x_test /= 255 # Normalize the testing data by dividing by 255
```

- حال نوبت به طراحی مدل میشود. یک مدل sequential داریم که شامل سه لایه است. لایه ورودی، میانی و خروجی. لایه ورودی از تابع فعال سازی RelU استفاده میکند و شامل 25 فیچر ورودی است. لایه میانی از همان تابع فعال سازی استفاده کرده ولی این بار شامل 32 نورون است. در انتها از softmax استفاده شد و 10 نورون خروجی داریم.

```
model = Sequential()
model.add(Dense(64, input_dim=25, activation='relu')) # input layer
model.add(Dense(32, activation='relu')) # hidden layer
model.add(Dense(10, activation='softmax')) # output layer
```

```
1 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                1664

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 4074 (15.91 KB)
Trainable params: 4074 (15.91 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- حال مدل را compile میکنیم. برای مسائل کلاس بندی از تابع ضرر categorical_crossentropy بهتر است استفاده کنیم و از تابع بهینه ساز adam چون بازدهی خوبی دارد استفاده میکنیم. و متریکی که میخواهیم طبق آن ارزیابی کنیم accuracy انتخاب شد.

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

- حال با دستور model.fit مدل را آموزش میدهیم:

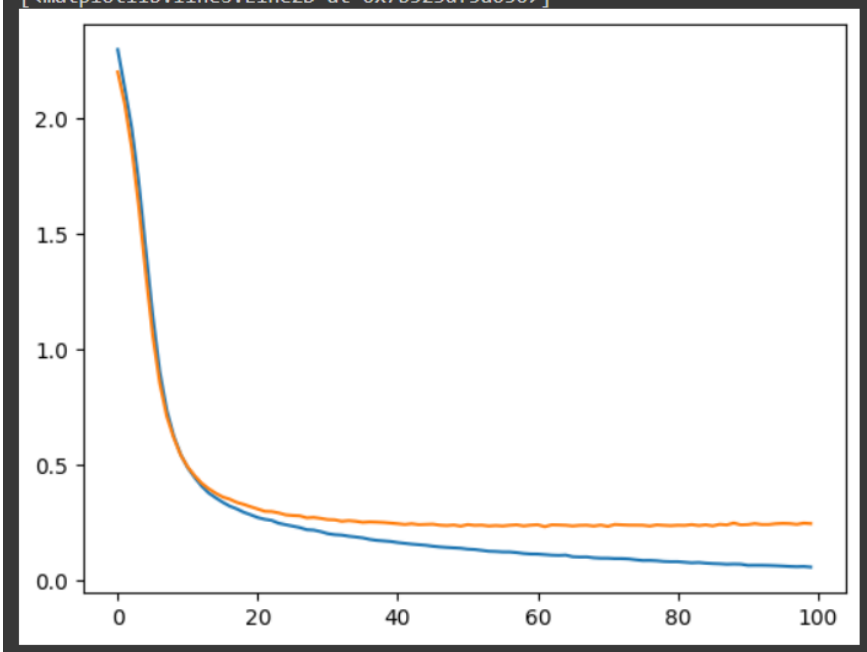
```
1 MLP_model = model.fit(x_train, y_train,
2           epochs=100,
3           batch_size=64, validation_data=(x_test, y_test))
```

- در اخر با استفاده از plot، ضررها و accuracy را نمایش میدهیم و همانطور که نشان داده شد، تابع ضرر رو به کاهش و accuracy رو به

افزایش است.(مقدار ضرر روی داده تست کمی بیشتر پس در نتیجه accuracy کمتری دارد.)

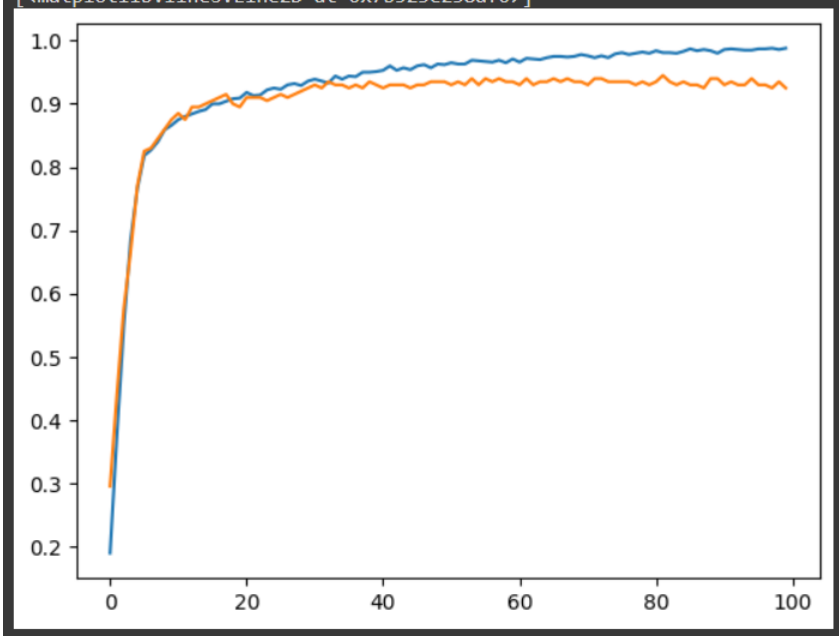
```
1 # plot Loss
2 plt.plot(MLP_model.history["loss"])
3 plt.plot(MLP_model.history["val_loss"])
```

```
[<matplotlib.lines.Line2D at 0x7b523df5a650>]
```



```
1 # plot accuracy
2 plt.plot(MLP_model.history["accuracy"])
3 plt.plot(MLP_model.history["val_accuracy"])
```

```
[<matplotlib.lines.Line2D at 0x7b523e238af0>]
```



سوال5: سوال از ما خواسته تا یک پرسپترون چند لایه تعریف کنیم که میتواند تابع XNOR را یاد بگیرد. پرسپترون طراحی شده شامل 3 لایه(ورودی، یک لایه میانی و یک لایه خروجی) است که به ترتیب 2 و 4 و 1 نورون دارند. لایه ورودی 2 نورون دارد زیرا دو عدد ورودی داریم و لایه خروجی 1 نورون دارد زیرا خروجی تابع یک عدد است. تعداد نورون های لایه میانی یک hyperparameter است که طبق تجربه میتواند تغییر کند.

حال مراحل و کد ها را شرح میدهم:

- ابتدا کتابخانه مورد نظر را صدا میکنیم:

Import the libraries:

```
[1]   1 import numpy as np
```

- سپس ورودی و خروجی تابع XNOR را تعریف میکنیم:

The XNOR function's input and target data

```
[2]   1 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
      2 Y = np.array([1, 0, 0, 1])
```

- در مرحله بعد، تعداد لایه ها و نورون ها و نرخ یادگیری و تعداد ایپاک های مدل را طراحی میکنیم:

The neural network architecture

```
[4]   1 input_size = 2
      2 hidden_size = 4
      3 output_size = 1
      4
      5 learning_rate = 0.1
      6 epochs = 10000
```

- سپس وزن ها و بایاس ها را بصورت رندوم، مقدار دهی اولیه میکنیم:

Initialize the weights and biases for each layer

```
[5]   1 input_layer_weights = np.random.uniform(size=(input_size, hidden_size))
      2 input_layer_bias = np.random.uniform(size=(1, hidden_size))
      3
      4 hidden_layer_weights = np.random.uniform(size=(hidden_size, output_size))
      5 hidden_layer_bias = np.random.uniform(size=(1, output_size))
```

- در این مدل، از تابع فعال ساز Sigmoid استفاده میشود. پس باید آن و مشتقش(که در فرآیند back propagation استفاده میشود) را تعریف کنیم:

The activation function and its derivative

```
[6]  1 def sigmoid(x):
     2     return 1 / (1 + np.exp(-x))
     3
     4 def sigmoid_derivative(x):
     5     return x * (1 - x)
```

- در نهایت، مدل را آموزش داده و نتیجه یادگیری آن را تست میکنیم:

Train the neural network

```
1 for epoch in range(epochs):
2     # Forward propagation
3     input_layer_output = sigmoid(np.dot(X, input_layer_weights) + input_layer_bias)
4     output_layer_output = sigmoid(np.dot(input_layer_output, hidden_layer_weights) + hidden_layer_bias)
5     error = Y.reshape(-1, 1) - output_layer_output
6
7     # Backpropagation
8     d_output = error * sigmoid_derivative(output_layer_output)
9     error_hidden_layer = d_output.dot(hidden_layer_weights.T)
10    d_hidden_layer = error_hidden_layer * sigmoid_derivative(input_layer_output)
11
12    # Updating the weights and biases
13    hidden_layer_weights += input_layer_output.T.dot(d_output) * learning_rate
14    hidden_layer_bias += np.sum(d_output, axis=0, keepdims=True) * learning_rate
15    input_layer_weights += X.T.dot(d_hidden_layer) * learning_rate
16    input_layer_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
17
18 # testing the neural network on XNOR inputs
19 predictions = (output_layer_output > 0.5).astype(int)
20 print("Predictions:", predictions)
```

همان طور که مشاهده میشود از فرمول x.W + b استفاده شده و خروجی این مقدار به تابع فعال ساز داده شده است(فرآیند forward propagation) سپس برای اپدیت کردن وزن ها و بایاس ها، از back propagation و مشتق تابع فعال سازی طبق تعریف استفاده شده است.(همه ی فرمول های استفاده شده طبق اسلاید درس میباشد!)

مقدار threshold برابر با 0.5 بوده است و این خروجی مدل آموزش دیده شده است:

```
Predictions: [[1]
 [0]
 [0]
 [1]]
```

سوال6: ابتدا کتابخانه ها مورد نیاز از جمله tensorflow, matplotlib, numpy را صدا میکنیم:

Libraries

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import tensorflow as tf
4 from tensorflow.keras.datasets import mnist # for MNIST dataset
5 from tensorflow.keras.models import Sequential
6 from tensorflow.keras.layers import Flatten, Dense
7 from tensorflow.keras.utils import to_categorical
```

سپس داده از روی این دیتاست لود کرده و پیش پردازش میکنیم(روش پیش پردازش عین همیشه است و داده های تست و آموزش به 255 تقسیم میشوند و همچنین تابع to_categorical برای تبدیل داده ها به one-hot vector است.):

Load and preprocess the MNIST **dataset**

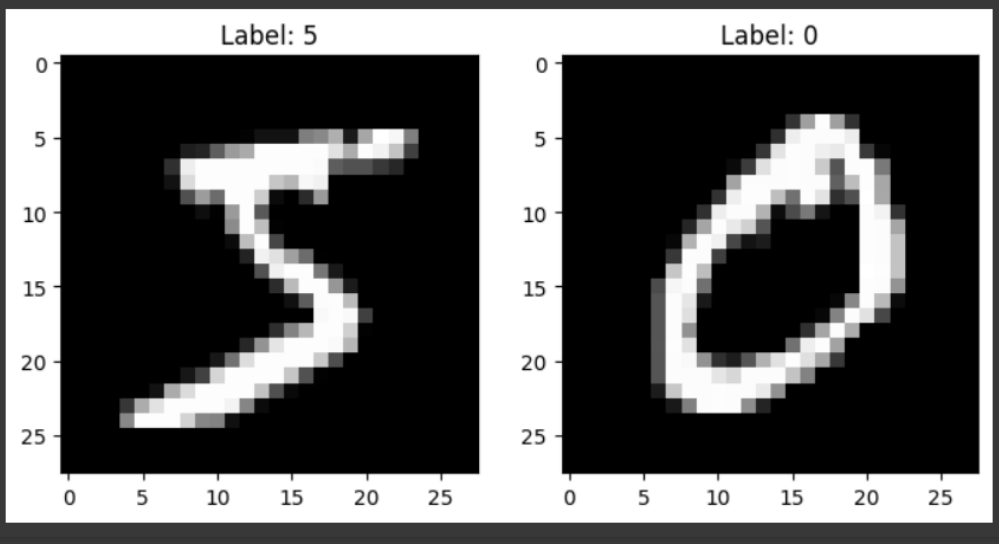
```
[4]  1 (x_train, y_train), (x_test, y_test) = mnist.load_data()
     2 x_train, x_test = x_train / 255.0, x_test / 255.0
     3 y_train, y_test = to_categorical(y_train), to_categorical(y_test)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
```

برای آشنایی بیشتر با این دیتاست، دو نمونه از داده آموزشی آن را رسم کردم:

Display 2 samples from the **dataset**

```
1 plt.figure(figsize=(8, 4))
2 for i in range(2):
3     plt.subplot(1, 2, i + 1)
4     plt.imshow(x_train[i], cmap='gray')
5     plt.title(f"Label: {np.argmax(y_train[i])}")
6 plt.show()
7
```



حال نوبت به تعریف مدل پیشنهادی میرسد. مدل پیشنهادی من شامل 4 لایه(2 لایه پنهان است) که به ترتیب لایه اول، لایه ورودی است که عکس ها را بعنوان ورودی میگیرد(این عکس ها 28*28 هستند) و لایه میانی اول 128 نورون دارد و تابع فعال سازی آن RelU است و لایه میانی دوم، 64

نورون دارد و تابع فعال سازی آن RelU است و در نهایت لایه خروجی، 10 نورون دارد(زیرا 10 کلاس یا 10 عدد داریم) و تابع فعال سازی آن Softmax است که در مسائل categorical استفاده میشود.

حال اگر علت این تعداد لایه و نورون هرکدام را بخواهیم شرح دهیم، میتوان گفت در لایه ورودی از flatten استفاده شده تا عکس های 28*28 به عکس های یک بعدی با 784 مقدار تبدیل شوند که این یک روش پیش پردازش است.

در لایه های پنهان ، از RelU بعنوان تابع فعال ساز استفاده شده زیرا یک تابع شناخته شده و بهینه برای مدل های پیچیده است.

تعداد نورون های لایه های میانی که به ترتیب 128 و 64 هستند، طبق سرچ و خواندن مدل های مختلف انتخاب شده که در طی سرچ، متوجه شدم که بصورت تجربی طبق پیچیدگی مدل، منابع و بهینه کردن hyperparameters تعداد نورون ها میتواند فرق کند و قانون خاصی برایش وجود ندارد(بهتر است توانی از 2 باشد). در این سوال، برای کسب حداقل دقت 95 درصد، این تعداد لایه و نورون جوابگو میباشد ولی اگر دقت خیلی بالاتر مدنظر باشد و یا مدل پیچیده تر باشد، تعداد لایه ها و نورون های آن باید بیشتر شوند.

Build the MLP **model**

```
[6]  1 model = Sequential()
     2 model.add(Flatten(input_shape=(28, 28)))  # Input layer (Flatten the 28x28 image)
     3 model.add(Dense(128, activation='relu'))   # Hidden layer with 128 neurons
     4 model.add(Dense(64, activation='relu'))    # Hidden layer with 64 neurons
     5 model.add(Dense(10, activation='softmax'))  # Output layer with 10 neurons (one for each digit)
```

حال که مدل خود را تعریف کردیم، آن را کامپایل کرده و آموزش میدهیم(تعداد ایپاک میتواند بیشتر باشد تا نتیجه بهتری گرفت):

Compile and **train** the model

```
 1 model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
 2 history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
 3

Epoch 1/10
1875/1875 [==============================] - 11s 5ms/step - loss: 0.2394 - accuracy: 0.9303 - val_loss: 0.1252 - val_accuracy: 0.9609
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1005 - accuracy: 0.9693 - val_loss: 0.1098 - val_accuracy: 0.9678
Epoch 3/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0713 - accuracy: 0.9770 - val_loss: 0.0821 - val_accuracy: 0.9736
Epoch 4/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0531 - accuracy: 0.9831 - val_loss: 0.0788 - val_accuracy: 0.9771
Epoch 5/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0416 - accuracy: 0.9869 - val_loss: 0.0721 - val_accuracy: 0.9787
Epoch 6/10
1875/1875 [==============================] - 14s 7ms/step - loss: 0.0343 - accuracy: 0.9887 - val_loss: 0.0829 - val_accuracy: 0.9761
Epoch 7/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0296 - accuracy: 0.9900 - val_loss: 0.0785 - val_accuracy: 0.9785
Epoch 8/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0231 - accuracy: 0.9923 - val_loss: 0.1005 - val_accuracy: 0.9743
Epoch 9/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0222 - accuracy: 0.9928 - val_loss: 0.0924 - val_accuracy: 0.9766
Epoch 10/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0183 - accuracy: 0.9938 - val_loss: 0.0892 - val_accuracy: 0.9809
```

حال میتوان نتیجه را روی داده تست ارزیابی کرد:

Evaluate the model on the test dataset

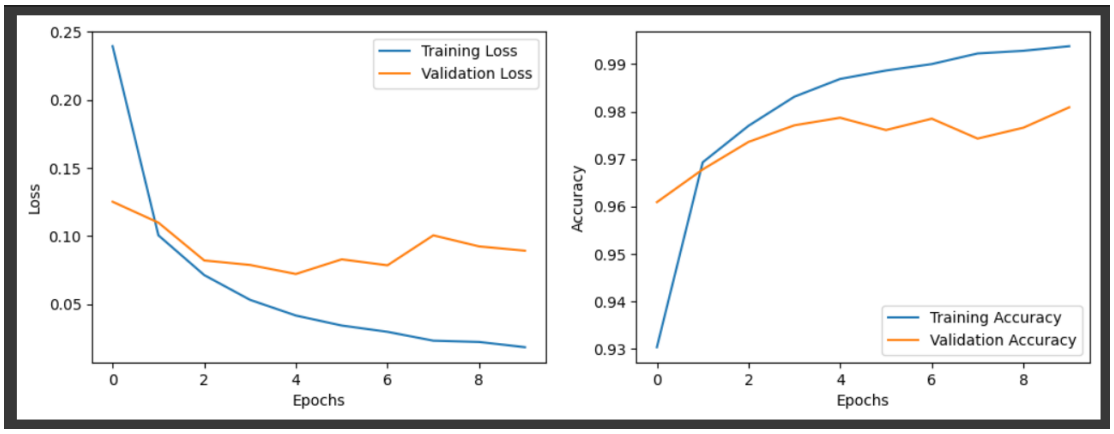
```
[9]  1 test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
     2
     3 print(f"Training Loss: {history.history['loss'][-1]:.4f}, Training Accuracy: {history.history['accuracy'][-1]:.4f}")
     4 print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}")

Training Loss: 0.0183, Training Accuracy: 0.9938
Test Loss: 0.0892, Test Accuracy: 0.9809
```

همان طور که مشاهده میشود دقت روی داده تست نزدیک به 99.4 درصد و روی داده تست حدود 98 درصد بوده است که خواسته مسئله را پوشش میدهد.

برای درک بهتر مقدار ضرر و دقت مدل ذکر شده، نمودار مربوط به آنها رسم شده است:

Plot loss and accuracy

```
[10]  1 plt.figure(figsize=(12, 4))
      2 plt.subplot(1, 2, 1)
      3 plt.plot(history.history['loss'], label='Training Loss')
      4 plt.plot(history.history['val_loss'], label='Validation Loss')
      5 plt.legend()
      6 plt.xlabel('Epochs')
      7 plt.ylabel('Loss')
      8
      9 plt.subplot(1, 2, 2)
     10 plt.plot(history.history['accuracy'], label='Training Accuracy')
     11 plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
     12 plt.legend()
     13 plt.xlabel('Epochs')
     14 plt.ylabel('Accuracy')
     15
     16 plt.show()
```



پایان