گزارش تمرین HW2 - محمد اصولیان

Q1)

پاسخ سوال در فایل پی دی اف در فایل زیپ میباشد.

Q2)

الف)

تابع فعالسازی خطی (Linear Activation Function) به صورت یک تابع خطی عمل می‌کند و خروجی آن با توجه به ورودی‌ها برابر با جمع خطی آن‌ها است. به عبارت دیگر، خروجی تابع فعالسازی خطی به صورت وزن‌دار از جمع ورودی‌ها بدست می‌آید. به عنوان مثال، تابع فعالسازی خطی معمولاً در لایه خروجی شبکه‌های عصبی استفاده می‌شود، زیرا به صورت مستقیم خروجی را تولید می‌کند.

تابع فعالسازی غیرخطی (Nonlinear Activation Function) با ورودی‌ها به صورت غیرخطی عمل می‌کند و خروجی غیرخطی تولید می‌کند. تابع‌های فعالسازی غیرخطی معروف شامل سیگموید (Sigmoid)، تانژانت هیپربولیک (Tanh)و رلو (ReLU) می‌شوند.

اصلی‌ترین تفاوت بین تابع فعالسازی خطی و غیرخطی در عملکرد آن‌ها است. تابع‌های فعالسازی غیرخطی نقش مهمی در توانایی شبکه‌های عصبی دارند که الگوهای پیچیده را یاد بگیرند و توانایی تقریب توابع غیرخطی را داشته باشند. در صورتی که از تابع فعالسازی خطی در شبکه چند لایه استفاده شود، شبکه همچنان خطی خواهد ماند و با افزایش تعداد لایه‌ها، قدرت انعطاف پذیری مدل افزایش پیدا نمی‌کند. استفاده از تابع فعالسازی غیرخطی در شبکه‌های عصبی به دلیل قابلیت آن‌ها در نمایش الگوهای پیچیده و افزایش ظرفیت شبکه معمولاً توصیه می‌شود.

ب)

در صورتی که در ابتدا وزن‌ها را صفر و بایاس را رندوم مقداردهی کنیم، فرایند یادگیری به درستی انجام نمی‌شود. در ابتدا با توجه به این که همه وزن‌ها صفر هستند، خروجی مدل برابر بایاس نورون‌های آخرین لایه می‌شود و چون بایاس ها رندوم هستند، خروجی مدل هم رندوم میباشد. در فرایند Backpropagation، با توجه به این که وزن‌ها صفر هستند، مشتق لاس نسبت به وزن‌ها هم صفر می‌شود و وزن‌ها تغییری نمی‌کنند. اما وزن‌های نورون‌ها لایه آخر تغییر کمی میکنند. در مراحل بعدی در هر مرحله تغییر وزن‌ها به یک لایه عقب تر می‌رسد، اما تغییرات وزن‌ها بسیار ناچیز و کوچک هستند. این مشکل باعث می‌شود که

1- یادگیری مدل بسیار کند انجام شود و در واقع در مراحل ابتدایی اصلا یادگیری‌ای انجام نمی‌شود.
2- مدل با احتمال زیادی در مینیمم محلی گیر کند.
3- در صورتی که بایاس‌ها هم صفر مقدار دهی شده باشند، مدل هیچ چیزی یاد نمیگیرد
4- در صورتی که بایاس ها مقدار یکسانی داشته باشند، وزن‌ها دقیقا یک فیچر را یاد میگیرند که باعث عملکر ضعیف مدل خواهد شد.

اما در صورتی که وزن‌ها را رندوم و بایاس را صفر مقدار دهی کنیم، پس از اولین back propagation، بایاس ها هم تغییر میکنند و به سمت مقدار ایده آل پیش میروند. صفر گذاشتن مقدار بایاس‌ها در صورتی که وزن ها رندوم مقدار دهی شده باشند، تاثیر منفی چندانی در عملکرد مدل نخواهد داشت. با این وجود پیشنهاد میشود که بایاس ها هم به صورت مقادیر رندوم کوچک در ابتدا مقداردهی کنیم.

ج)

از دیدگاه قابلیت تعمیم، شبکه‌های عصبی عمیق (Deep Neural Networks) دارای قابلیت کمتری نسبت به شبکه‌های عصبی سطح پایین‌تر مانند شبکه‌های عصبی ساده (Single-Layer Perceptron) یا شبکه‌های عصبی پرسپترون چندلایه (Multi-Layer Perceptron)هستند. این امر به دلیل تعداد بالای پارامترها و لایه‌های شبکه عصبی عمیق است که باعث می‌شود شبکه به راحتی بتواند الگوهای پیچیده را یاد بگیرد و در نتیجه برروی داده‌های آموزشی عملکرد خوبی داشته باشد. اما در برخی موارد، این افزایش پیچیدگی ممکن است باعث شود شبکه بیش‌برازش (Overfit) شود و عملکرد ضعیفی روی داده‌های جدید و ناشناخته داشته باشد.

در مقابل، شبکه‌های عصبی سطح پایین‌تر معمولاً تعداد کمتری پارامتر و لایه دارند و از این رو قابلیت تعمیم بیشتری نسبت به شبکه‌های عصبی عمیق دارند. این نوع شبکه‌ها معمولاً در مواقعی که داده‌ها به صورت ساده و قابل جداسازی باشند، به خوبی عمل می‌کنند. به عنوان مثال، در مسائلی که داده‌ها به صورت خطی قابل جداسازی باشند، شبکه‌های عصبی ساده مانند شبکه‌های عصبی ساده یا پرسپترون چندلایه به خوبی عملکرد می‌کنند و قابلیت تعمیم بالایی دارند.

بنابراین، در کل، شبکه‌های عصبی سطح پایین‌تر معمولاً قابلیت تعمیم بیشتری دارند، در حالی که شبکه‌های عصبی عمیق به دلیل قابلیت یادگیری الگوهای پیچیده، ممکن است قابلیت تعمیم کمتری نسبت به شبکه‌های سطح پایین‌تر داشته باشند. با این حال، با استفاده از روش‌های مناسب برای کنترل بیش‌برازش، می‌توان تا حد زیادی قابلیت تعمیم شبکه‌های عصبی عمیق را بهبود داد.

در نهایت می‌توان گفت که بین پیچیدگی مدل و قدرت یادگیری داده‌های train، و قابلیت تعمیم مدل یک tradeoff وجود دارد. باید با توجه به مسئله و دقت مورد نیاز انتخاب کرد که از کدام مدل بهتر است استفاده کرد.

د)

مزایا:

سرعت همگرایی بالا: استفاده از مشتق دوم می‌تواند منجر به سرعت همگرایی بالاتر در فرآیند آموزش شود. با در نظر گرفتن مشتق دوم، می‌توان تغییرات جزئی در توابع هدف را تشخیص داد و در نتیجه، مسیر بهینه‌تری برای به‌روزرسانی وزن‌ها انتخاب کرد.

موثر برای تغییرات شدید: استفاده از مشتق دوم در فرآیند آموزش مدل می‌تواند بهبود قابل توجهی در تطابق مدل با داده‌های آموزشی در مواجهه با تغییرات شدید داشته باشد. با توجه به اطلاعات بیشتر در مورد منحنی‌های تابع هدف، می‌توان از تغییرات دقیق‌تری در وزن‌ها بهره برد.

معایب:

محاسبات پیچیده: محاسبه و استفاده از مشتق دوم معمولاً محاسبات پیچیده‌تر و هزینه‌برتر را می‌طلبد. به خصوص در شبکه‌های عصبی با تعداد پارامترهای زیاد، محاسبه مشتق دوم هزینه‌بر و زمان‌بر می‌شود. این محاسبات ممکن است محدودیت‌های حافظه و زمان را ایجاد کرده و مشکلاتی در مقیاس‌پذیری را ایجاد کند.

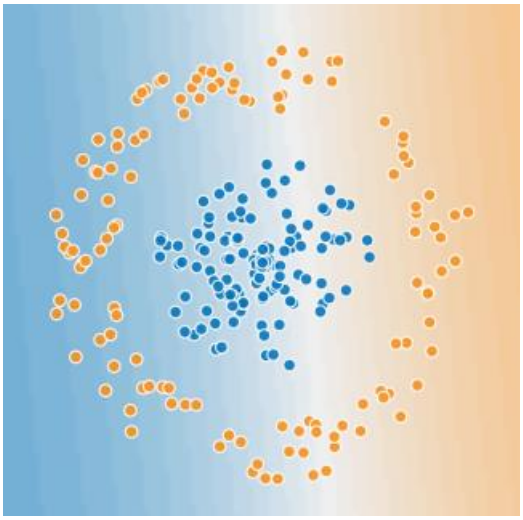
حساسیت به نقاط شده‌ی محلی :(Local Minima) مشتق دوم نقاط شده‌ی محلی در فضای جستجو را نیز در نظر می‌گیرد و این ممکن است به مشکلاتی در همگرایی به جواب بهینه منجر شود. در شبکه‌های عصبی با تعداد بالای پارامترها، وجود نقاط شده‌ی محلی ممکن است رخ دهد و موجب از دست دادن جواب بهینه یا حتی بیش‌برازش شود.

پیچیدگی محاسباتی در مقیاس بزرگ: استفاده از مشتق دوم در شبکه‌های عصبی بزرگ می‌تواند به مشکلات پیچیدگی محاسباتی منجر شود. این مشکلات شامل محدودیت‌های حافظه، زمان محاسباتی بالا و مشکلات بهینه‌سازی کلی می‌شود.

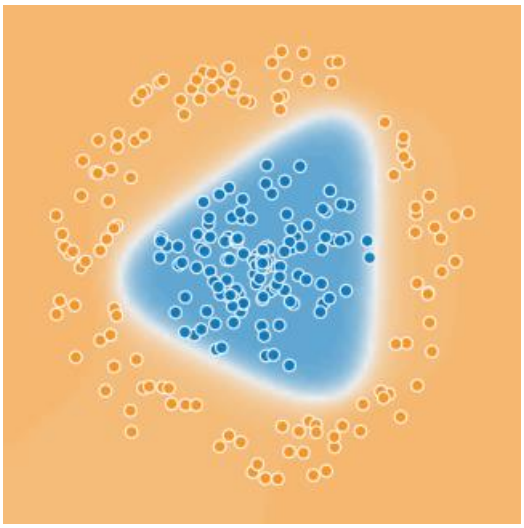
در کل، استفاده از مشتق دوم به جای مشتق اول برای آپدیت وزن‌ها مزایا و معایب خاص خود را دارد. از طرفی، استفاده از مشتق دوم می‌تواند سرعت همگرایی بالاتری را در فرآیند آموزش مدل‌ها فراهم کند و منجر به بهبود تطابق مدل با داده‌های آموزشی در مواجهه با تغییرات شدید شود. از طرف دیگر، استفاده از مشتق دوم نیازمند محاسبات پیچیده‌تر و زمان‌برتر است و می‌تواند به محدودیت‌های حافظه و زمان محاسباتی منجر شود. همچنین، حساسیت به نقاط شده‌ی محلی و پیچیدگی محاسباتی در مقیاس بزرگ نیز از معایب استفاده از مشتق دوم است. در نهایت، استفاده از مشتق دوم به جای مشتق اول برای آپدیت وزن‌ها بستگی به مسئله و الگوریتم بهینه‌سازی مورد استفاده دارد و باید به دقت و با توجه به محدودیت‌های محاسباتی و مقیاس مسئله مورد بررسی قرار گیرد.

(Q3

(Dataset1

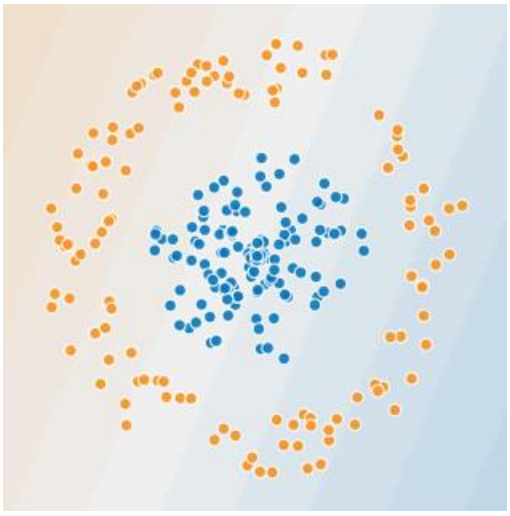


برای این دیتا ست تابع tanh به خوبی دو کلاس را از هم جدا کرد:



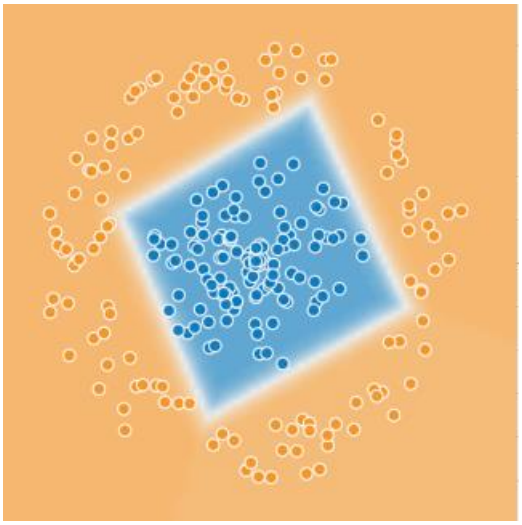
تابع سیگموید هم مشابه تابع tanh این دو دسته را جدا کرد اما تفاوت بین این دو این بود که تابع tanh خیلی زود تر از سیگموید همگرا شد و مدل سریع تر دسته بندی را یاد گرفت.

تابع خطی نتوانست دسته بندی را به درستی انجام دهد:



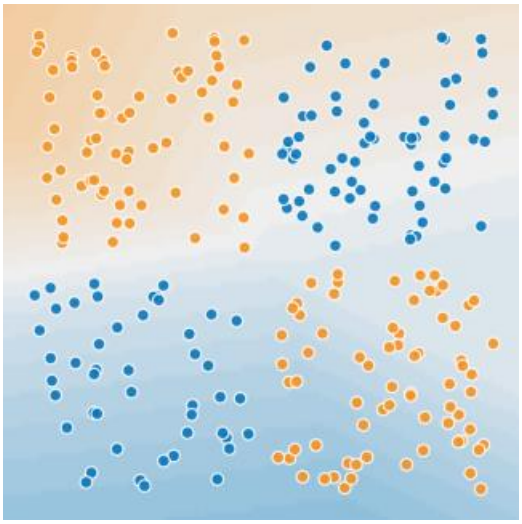
بلکه تنها با نزدیک کردن وزن ها به صفر، سعی کرد تا از بیشتر شدن خطا جلوگیری کند. یعنی به جای این دسته بندی را انجام دهد تنها سعی کرد که از انتخاب اشتباه دوری کند.

تابع ریلو هم دسته بندی را انجام داد اما با استفاده از خط های شکسته:

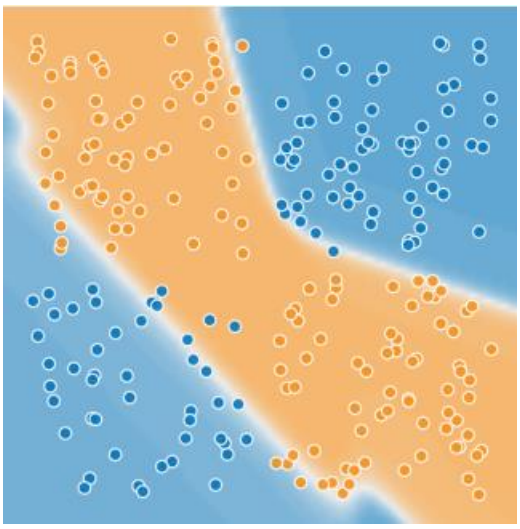


نتیجه tanh و sigmoid از ریلو بهتر بود اما مزیتی که ریلو در ابعاد بزرگ دیتاست‌ها دارد، سرعت اجرای بیشتر است که در این دیتاست کوچک احساس نمیشود.

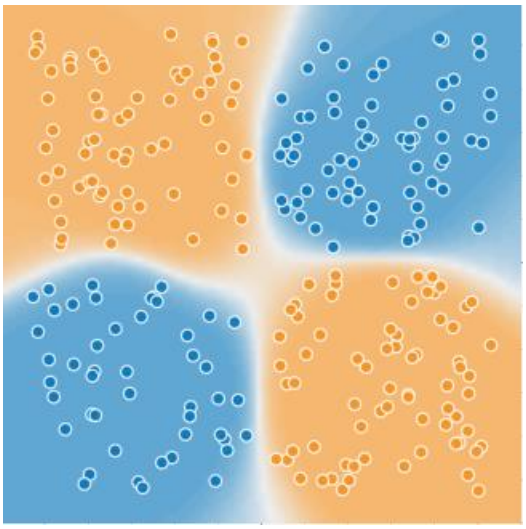
(Dataset2



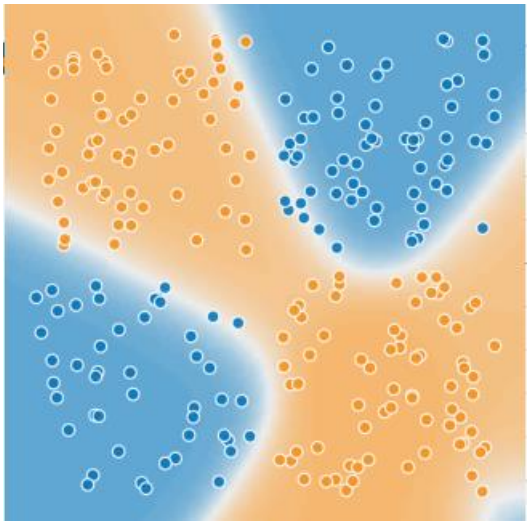
تابع tanh دسته بندی را به این صورت انجام داد:



دسته بندی هنوز هم جای بهتر شدن دارد. اما به علت این که لایه میانی تنها 3 نورون دارد، انعطاف پذیری مدل کم است. اگر تعداد نورون ها را بیشتر کنیم دسته بندی دقیق تری انجام میدهد. مثلا در شکل زیر از 5 نورون استفاده شده:

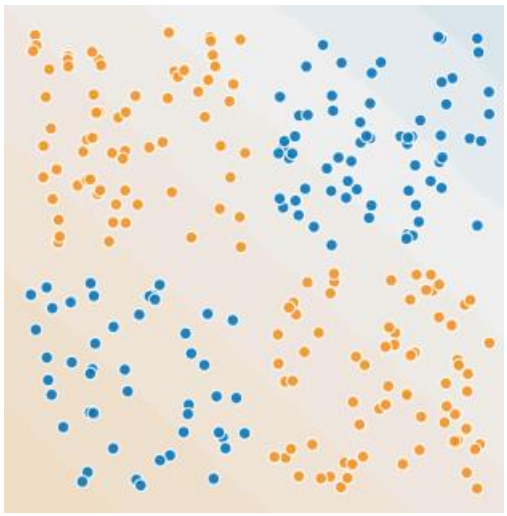


تابع سیگموید هم به این صورت دسته بندی را انجام داد:

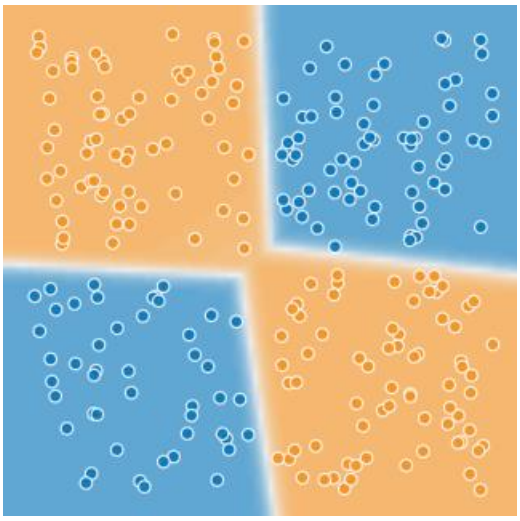


این تابع هم شبیه تابع tanh عمل کرده. اگر به جای 3 نورون از 5 نورون استفاده میکردیم دسته بندی خیلی بهتری انجام میداد. مجددا تفاوت دو تابع این است که tanh خیلی سریع تر از sigmoid همگرا میشود.

تابع خطی مانند دیتاست قبلی عملکرده و به جای انجام دسته بندی سعی در صفر کردن وزن ها داشته تا از خطا تا حد ممکن دوری کند:



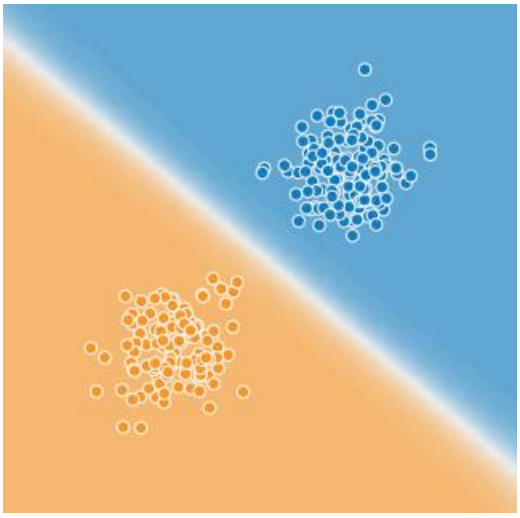
تابع ریلو با 3 نورون هم توانسته دسته بندی خوبی را انجام دهد:



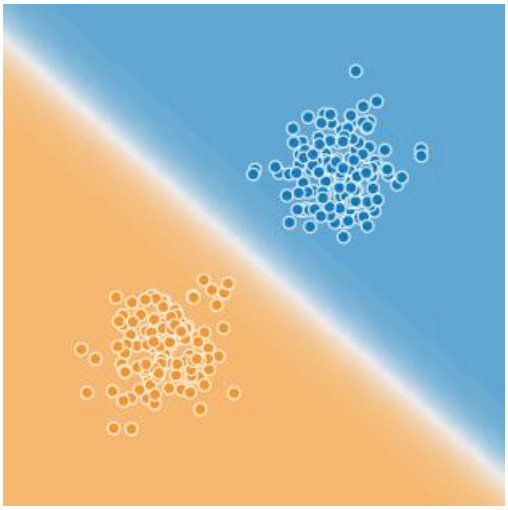
Dataset3)

این دیتاست بسیار ساده است و تنها با یک خط هم حل میشود بنابراین تمام توابع عملکرد خوبی را داشته اند:

Tanh:

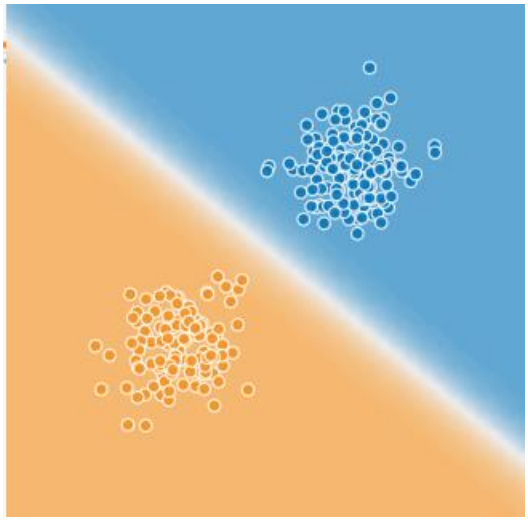


Sigmoid:



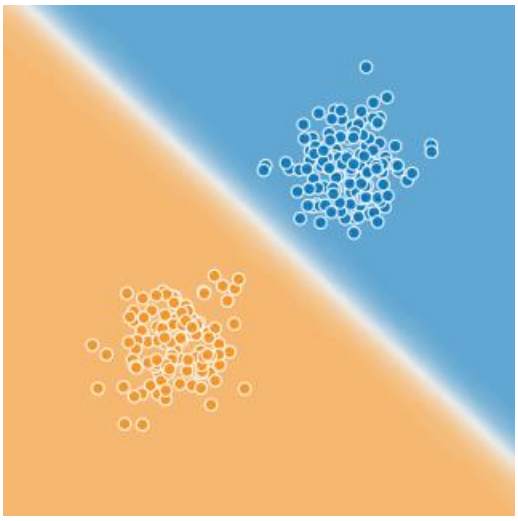
با وجود ساده بودن دیتاست، همچنان کند تر بودن سیگموید نسبت به tanh مشهود است.

Linear:



تابع خطی این بار به خوبی عملکرده چون برای حل این مسئله به چیزی بیشتر از یک خط هم نیاز نیست.

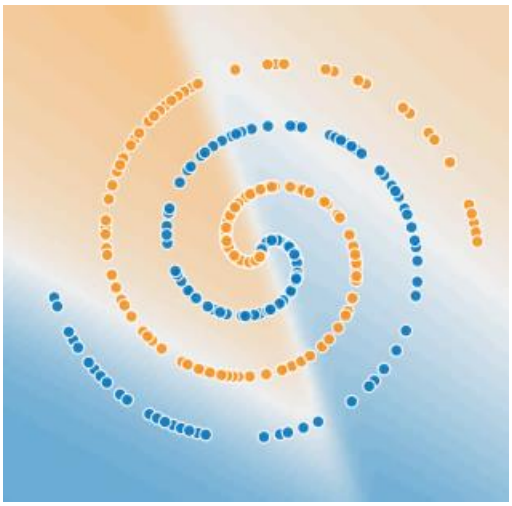
Relu:



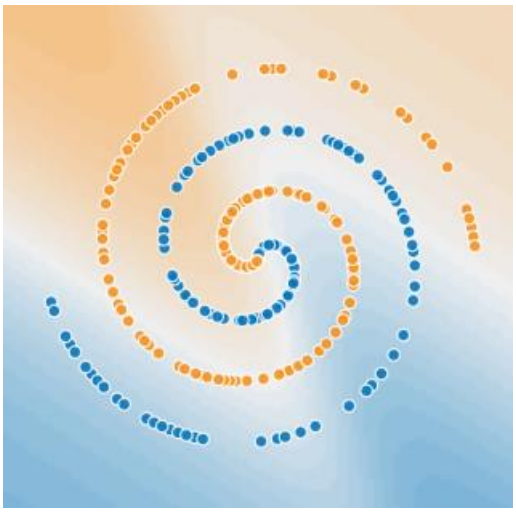
Dataset4)

این دیتاست از همه پیچیده تر است و هیچ کدام از توابع عملکرد خوبی نداشته اند.

تابع tanh:



Sigmoid:

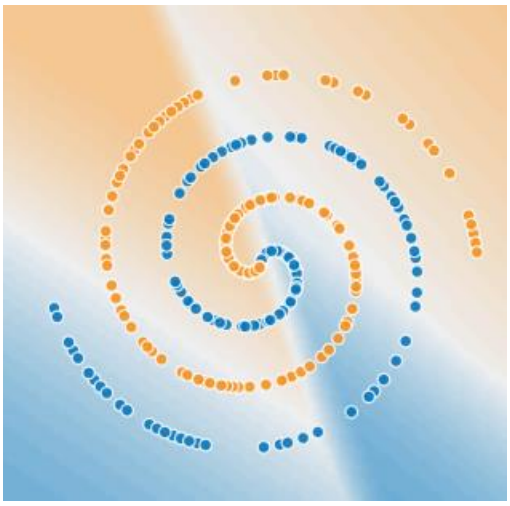


سیگموید هم به شکلی مشابه tanh رسیده. اما چندین دفعه اول به جای رسیدن به این دسته بندی، در یک لوکال مینیمم گیر می افتاد. پس از 4 بار اجرا کردن، سیگموید توانست در مرحله 500 ام به این شکل برسد.

Linear:

تابع خطی نمیتواند چنین دسته بندی ای انجام دهد. اما به طور حدودی میتوان گفت که جهت نقاط آبی را درست پیدا کرده.

Relu:



ریلو هم به شکلی مشابه tanh و سیگموید رسید، اما با خطوط شکسته.

در کل میتوان گفت که تابع خطی انعطاف پذیری بسیار پایینی دارد. به همین علت نمیتواند دسته بندی مناسبی را انجام دهد. توابع دیگر انعطاف پذیری بیشتری دارند.

بین tanh و sigmoid، tanh عملکرد بهتری داشته. زیرا هم کمتر در لوکال مینیمم می افتاد هم سریع تر همگرا میشد.

تابع Relu هم انعطاف خوبی داشت و در اکثر موارد مدلی مشابه tanh و sigmoid ارائه میداد با این تفاوت که برای دسته بندی به جای خطوط خمیده از خطوط شکسته استفاده میکرد.

Q4)

برای این سوال طبق بلوک‌های کد پیش رفتم. برای حل این سوال، نوتبوک را در کولب آپلود و اجرا کردم.

ابتدا با اجرای این بلوک، داده ها را دانلود کردم:

```
!gdown --fuzzy https://drive.google.com/file/d/1QJrQsEYOfPBn1LoIeYMZ2HFBRC0AY-6F/view?usp=sharing
!gdown --fuzzy https://drive.google.com/file/d/1zStcaVl_34RrYIfVObuM4xzB6s8xwvBi/view?usp=sharing
```

```
Downloading...
From: https://drive.google.com/uc?id=1QJrQsEYOfPBn1LoIeYMZ2HFBRC0AY-6F
To: /content/dataset.py
100% 909/909 [00:00<00:00, 3.00MB/s]
Downloading...
From: https://drive.google.com/uc?id=1zStcaVl_34RrYIfVObuM4xzB6s8xwvBi
To: /content/Data_hoda_full.mat
100% 3.99M/3.99M [00:00<00:00, 170MB/s]
```

این دیتاست، مربوط به اعداد دست نویس فارسی میباشد. تصاویر به صورت 5*5 هستند و 1200 داده دارد که 1000 تای آنها برای train و 200 تای دیگر برای تست استفاده می‌شود.

با استفاده از این تابع که مربوط به کتابخانه hoda میشود، داده ها را در متغیر لود میکنیم:

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

در این قسمت فرمت داده‌های تست و ترین را به صورت آرایه نامپای در میآوریم. همچنین خروجی را به شکل one-hot در میآوریم:

```
# 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)
```

میبینیم که تغییرات مورد نظر اعمال شده و ابعاد لیبل ها به صورت 10*1000 و 10*200 در آمده است.

```
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.]
```

در این قسمت نوع اعداد ورودی را به float تغییر میدهیم و همچنین نرمال سازی میکنیم. این کار باعث میشود که رینج اعداد ورودی، به جای 0 تا 255، از 0 تا 1 باشد که به train بهتر مدل کمک میکند.

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

سپس مدل را ابتدا ، تنها با یک لایه ورودی و یک لایه خروجی و بدون لایه مخفی ایجاد کردم. با توجه به این که مسئله ما دسته بندی بندی ده کلاسه بود، برای لایه آخر از تابع فعالسازی softmax استفاده کردیم تا خروجی نورون ها را به شکل احتمال در بیاورد.

```
model = Sequential([
    keras.layers.InputLayer(25),
    Dense(10, activation='softmax'),
])
```

سامری مدل به این صورت است که نشان میدهد مدل به درستی ایجاد شده:

```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 10)                260

=================================================================
Total params: 260 (1.02 KB)
Trainable params: 260 (1.02 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

در این قسمت مدل را کامپایل میکنیم. از adam به عنوان optimizer و از cross entropy به عنوان loss استفاده میکنیم. همچنین مطابق صورت سوال، accuracy را به عنوان متریک تنظیم میکنیم:

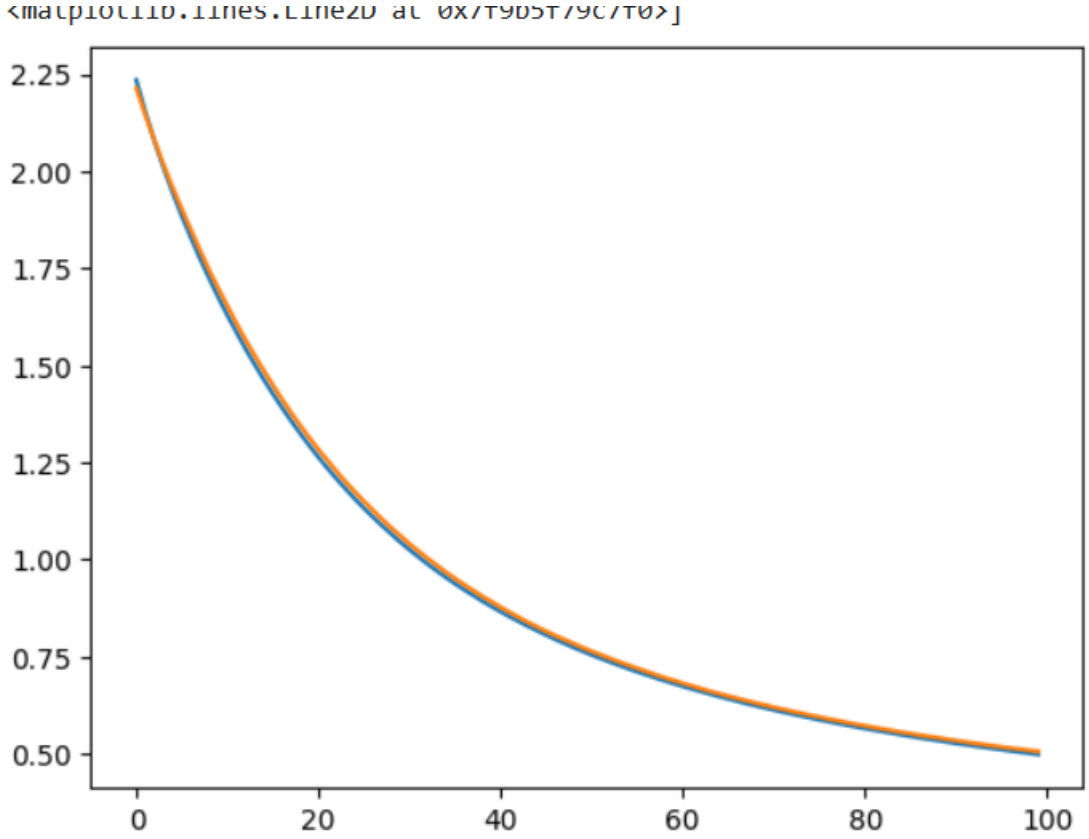
```
###################################
model.compile(optimizer='adam', loss='CategoricalCrossentropy', metrics='accuracy')
```

سپس مدل را با داده هایی که در اختیار داریم، فیت میکنیم:

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

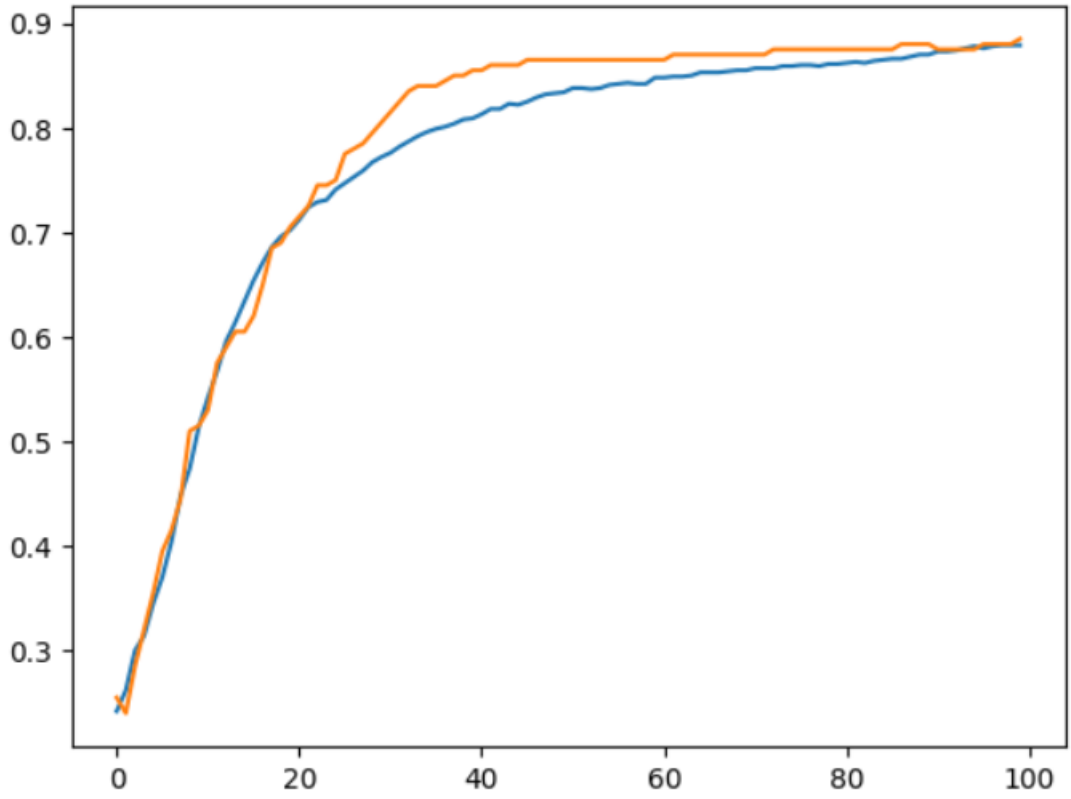
```
Epoch 72/100
```

لاس مدل طی 100 epoch به این صورت است:



میبنیم که روند یادگیری مدل به خوبی انجام شده و مقدار لاس، به مرور کاهش میابد. همچنین تفاوت بین train و validation بسیار کم است که به معنای این است که مدل، اورفیت نشده است.

همچنین دقت مدل طی 100 ایپاک به این صورت است:



که دقت به خوبی تا 87 درصد افزایش یافته و تفاوت کم بین validation و train نشان میدهد که مدل اورفیت نشده است.

این بار ساختار مدل را عوض کردم و یک لایه مخفی به آن اضافه کردم تا تغییرات مدل را بررسی کنیم. در لایه مخفی 15 نورون قرار میدهیم تا تعداد نورونهای لایه ها از زیاد به کم باشد.

```python
# In this Create the model, input dim=25 and output dim = 10

###################################
# you code here
###################################
model = Sequential([
    keras.layers.InputLayer(25),
    keras.layers.Dense(15, activation='relu'),
    Dense(10, activation='softmax'),
])
```

```python
model.summary()
```

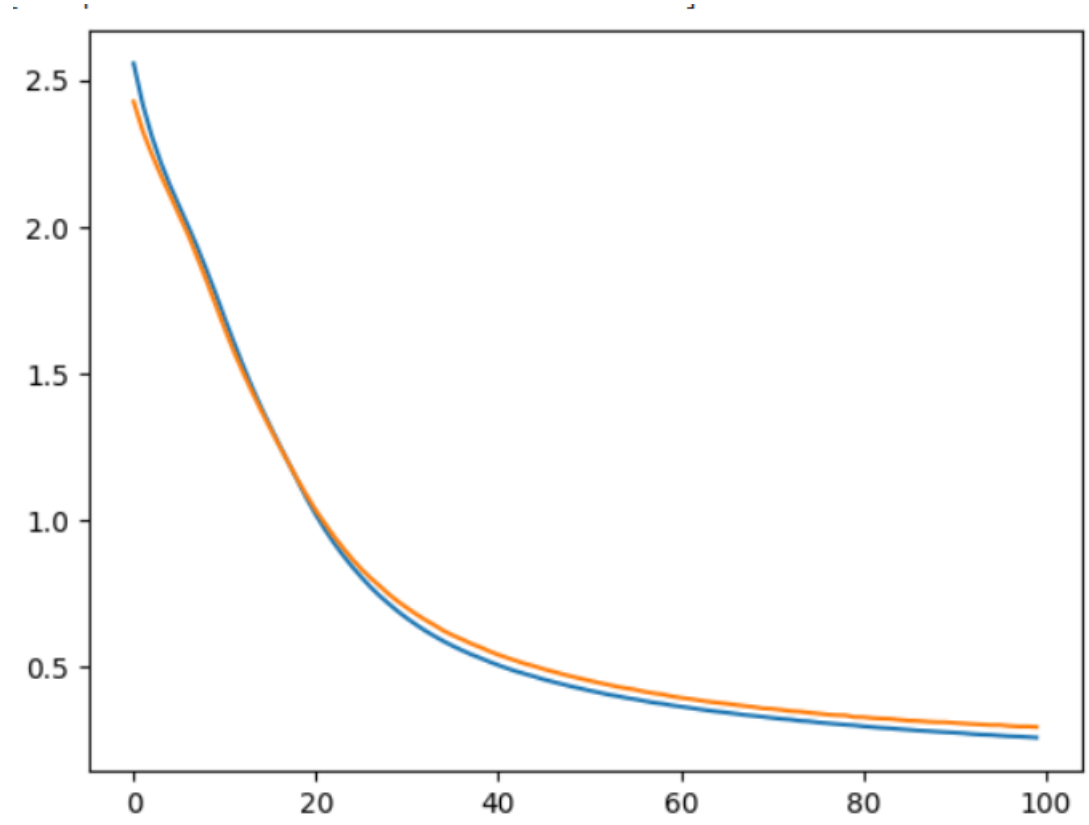
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 15)                390

 dense_4 (Dense)             (None, 10)                160

=================================================================
Total params: 550 (2.15 KB)
Trainable params: 550 (2.15 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

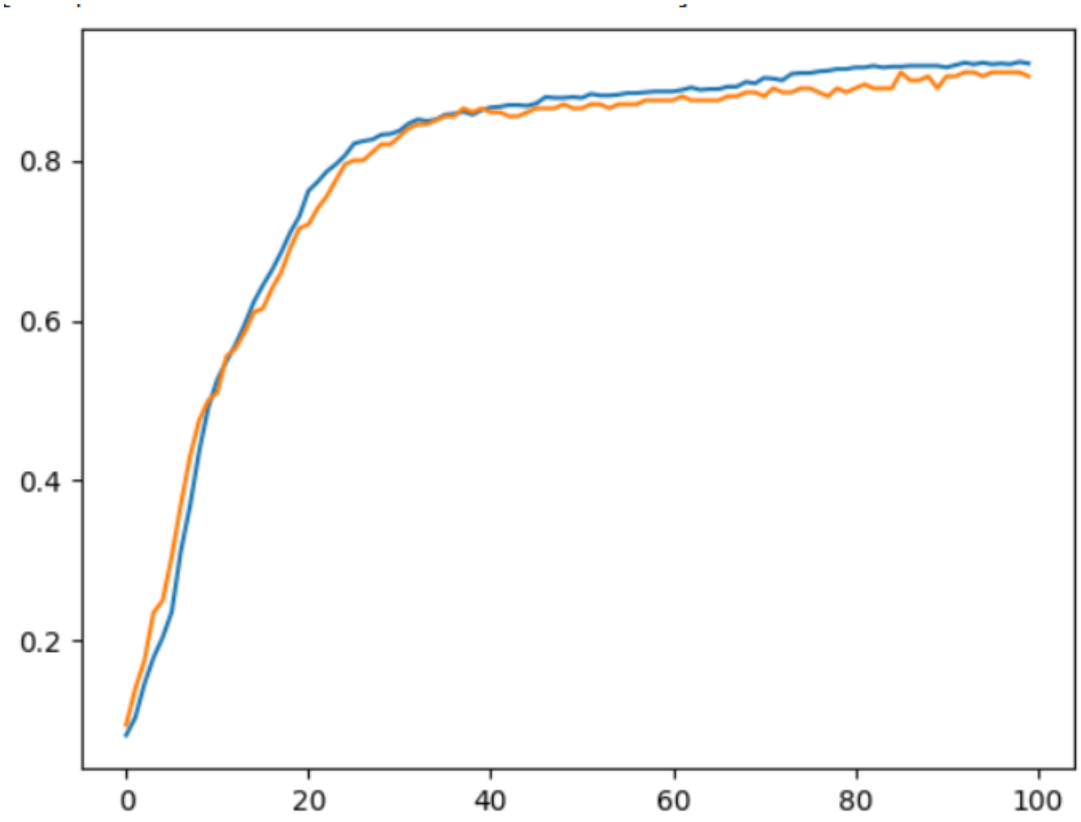
سایر هایپرپارامترهای مدل مثل اپتیمایزر و اکتیویشن فانکشن لایه اخر نیازی به تغییر ندارند.

حالا مدل را ترین میکنیم:

```
Epoch 100/100
16/16 [==============================] - 0s 5ms/step - loss: 0.2594 - accuracy: 0.9210 - val_loss: 0.2957 - val_accuracy: 0.9050
```

میبینیم که دقت مدل در ترین و تست حدود 3 درصد افزایش پیدا کرده. همچنین نمودار های لاس و دقت به این صورت هستند:


2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100


0.8
0.6
0.4
0.2
0
20
40
60
80
100

با توجه به نمودار میبینیم که مقدار اورفیتیگ از مدل قبلی هم کم تر شده زیرا دقت و لاس نمونه های ترین و تست بسیار نزدیک به هم هستند.

بار دیگر از تعداد بیشتری لایه های مخفی استفاده میکنیم. این بار از 3 لایه مخفی استفاده میکنیم و تعداد نورون های لایه های مخفی را به ترتیب 50 و 30 و 15 قرار میدهیم.

```
# In this Create the model, input dim=25 and output dim = 10

###################################
# you code here
###################################
model = Sequential([
    keras.layers.InputLayer(25),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(30, activation='relu'),
    keras.layers.Dense(15, activation='relu'),
    Dense(10, activation='softmax'),
])
```

```
model.summary()
```

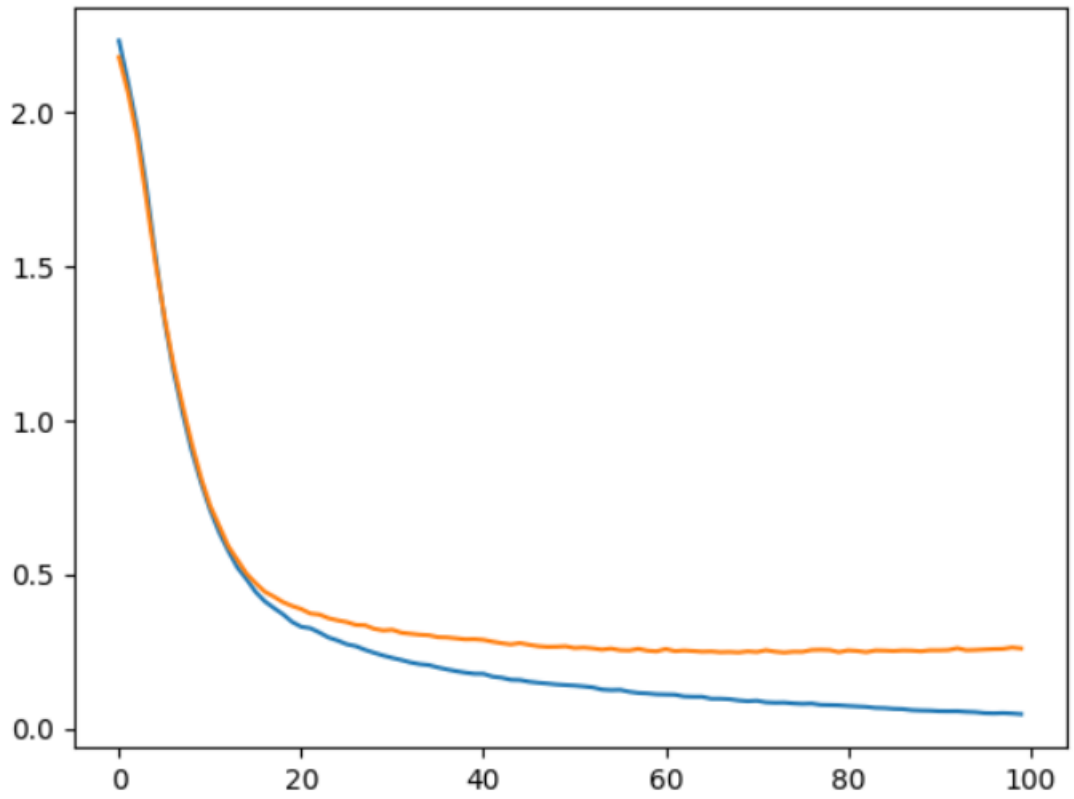
```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 50)                1300

 dense_10 (Dense)            (None, 30)                1530

 dense_11 (Dense)            (None, 15)                465

 dense_12 (Dense)            (None, 10)                160

=================================================================
Total params: 3455 (13.50 KB)
Trainable params: 3455 (13.50 KB)
Non-trainable params: 0 (0.00 Byte)
```

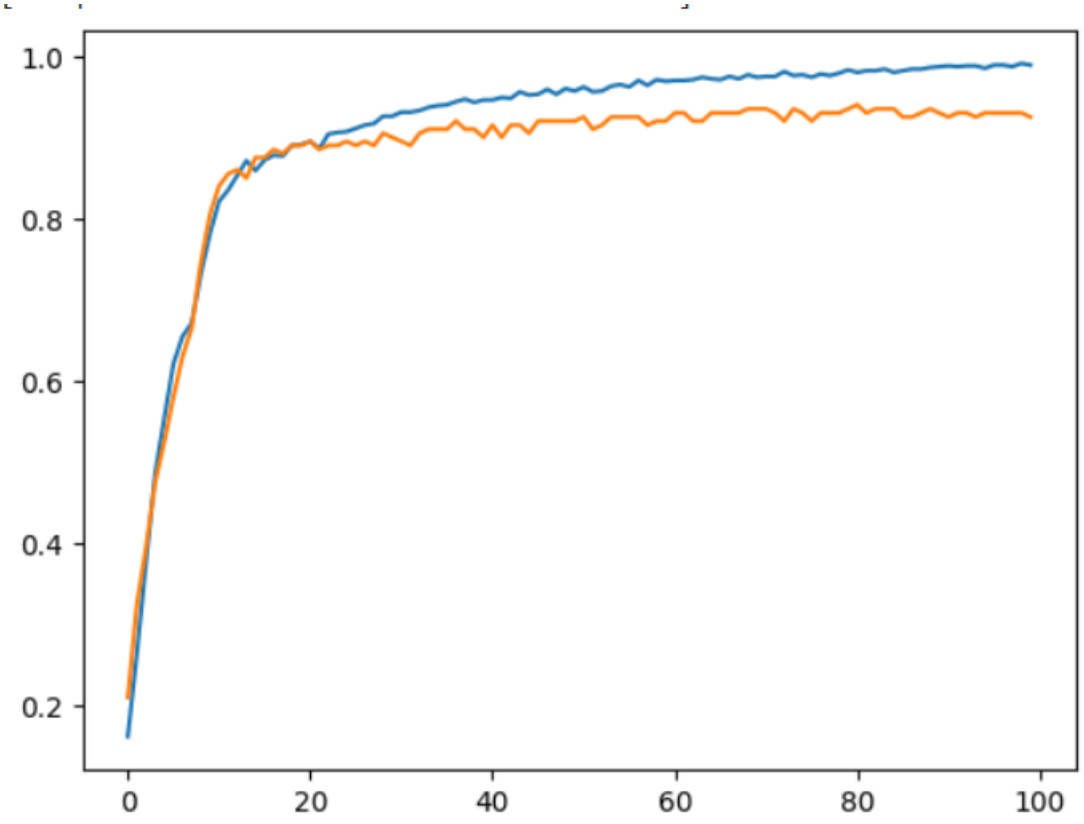
البته این احتمال وجود دارد که این بار به خاطر تعداد زیاد پارامترها، مدل آندرفیت شود و ترین به درستی در این تعداد مرحله صورت نگیرد.

حالا مدل را ترین کرده و نتایج را بررسی میکنیم:

```
Epoch 99/100
16/16 [==============================] - 0s 5ms/step - loss: 0.0480 - accuracy: 0.9910 - val_loss: 0.2620 - val_accuracy: 0.9300
Epoch 100/100
16/16 [==============================] - 0s 4ms/step - loss: 0.0459 - accuracy: 0.9890 - val_loss: 0.2590 - val_accuracy: 0.9250
```

میبینیم که دقت مدل در داده های ترین بهبود قابل توجه داشته و به 98 درصد رسیده. دقت در داده های تست هم دو درصد بهبود داشته اما به علت پیچیدگی زیاد نورون، مدل اورفیت شده و دقت در تست و ترین فاصله زیادی از هم گرفته:





البته با وجود این که مدل اورفیت شده، باز هم دقت آن در داده های تست بهتر از دو مدل دیگر است. مشکل اور فیتینگ را میتوان با افزایش داده ها حل کرد.

Q5)

برای حل این سوال ابتدا مدل مناسب را انتخاب کردم. با مدل هایی که در سوال 3 تست کردم، متوجه شدم که مدلی با یک لایه میانی با 3 نورون و با اکتیویشن فانکشن ریلو میتواند گزینه مناسبی باشد. بنابراین مدل را به این صورت طراحی کردم:

```
#
#
#   O     O
#         O       O
#   O     O
#   X  W1 b1 W2  b2  Y
#
```

ابتدا دیتاست را ایجاد و لیبل‌ها را تعیین کردم:

```
[72] x_train = np.array([[[0,0]], [[0,1]], [[1,0]], [[1,1]]])
     y_train = np.array([1, 0, 0, 1])
```

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

```
(4, 1, 2)
(4,)
```

علت سه بعدی بودن x_train این است که هر نمونه را یک batch در نظر گرفتم. در واقع بعد سوم همان batch است. لیبل ها هم همان تابع xnor هستند.

سپس با توجه به شکل مدل که در بالا نشان دادم، وزن ها و بایاس ها را به صور رندوم مقداردهی کردم:

```
] W1 = np.random.rand(2, 3)
  W2 = np.random.rand(3, 1)
  b1 = np.random.rand(1,3)
  b2 = np.random.rand(1,1)
```

ابعاد وزن ها به این صورت میباشد:

```
[76] print(f'{W1.shape=} {W2.shape=} {b1.shape=} {b2.shape=}')
```

```
W1.shape=(2, 3) W2.shape=(3, 1) b1.shape=(1, 3) b2.shape=(1, 1)
```

سپس توابع مورد نیازم را تعریف کردم:

```
[90] def ReLU(a):
       return np.maximum(a, 0)

     def ReLU_derivitive(a):
       return a > 0

     def sigmoid(a):
       return 1 / (1 + np.exp(-1 * a))

     def BinaryCrossEntropy(yhat, y):
       return -1*(y*np.log2(yhat) + (1-y)*np.log2(1-yhat))
```

تابع ریلو برای اکتیویشن لایه مخفی و مشتق آن برای backpropagation، تابع sigmoid برای اکتیویشن لایه آخر(چون دسته بندی دو کلاسه است) و تابع BinaryCrossEntropy برای محاسبه loss.

مسیر forward را به این صورت تعیین کردم:

```
def forward(x, W1, W2, b1, b2):
  z1 = np.dot(x, W1) + b1
  a1 = ReLU(z1)
  z2 = np.dot(a1, W2) + b2
  a2 = sigmoid(z2)
  yhat = a2
  return yhat
```

تابع back propagation را با کمک اینترنت و محاسبه مشتقات تکمیل کردم:

```
def backprop(x, W1, W2, b1, b2, y, alpha):
  z1 = np.dot(x, W1) + b1
  a1 = ReLU(z1)
  z2 = np.dot(a1, W2) + b2
  a2 = sigmoid(z2)

  dl_dz2 = a2 - y
  dl_db2 = dl_dz2
  dl_dW2 = np.dot(a1.T, dl_dz2)

  dl_da1 = np.dot(W2, dl_dz2)
  dl_dz1 = dl_da1 * ReLU_derivitive(z1).T

  dl_db1 = dl_dz1
  dl_dW1 = np.dot(dl_dz1, x).T

  b1 = b1 - alpha * dl_db1.T
  W1 = W1 - alpha * dl_dW1
  b2 = b2 - alpha * dl_db2
  W2 = W2 - alpha * dl_dW2

  return W1, W2, b1, b2
```

حالا تمام ابزار مورد نیاز برای آموزش یک شبکه عصبی را داریم. قبل از آموزش، پیشبینی مدل برای چهارنمونه را ببینیم:

```
print('Before Learning')
print(forward(x_train[0], W1, W2, b1, b2))   # x = 0, 0    y = 1
print(forward(x_train[1], W1, W2, b1, b2))   # x = 0, 1    y = 0
print(forward(x_train[2], W1, W2, b1, b2))   # x = 1, 0    y = 0
print(forward(x_train[3], W1, W2, b1, b2))   # x = 1, 1    y = 1
```

```
Before Learning
[[0.7819563]]
[[0.83897195]]
[[0.88531668]]
[[0.91813449]]
```

میبینیم که پیشبینی به صورت رندوم انجام می‌شود و متناسب با لیبل‌ها نیست.

حالا با دو حلقه تو در تو، عملیات train را انجام میدهیم. قبل از شروع حلقه مقادیر alpha و تعداد ایپاک ها را مشخص میکنیم.

```
alpha = 0.1
epochs = 500

for epoch in range(epochs):
  for i in range(4):
    yhat = forward(x_train[i], W1, W2, b1, b2)
    W1, W2, b1, b2 = backprop(x_train[i], W1, W2, b1, b2, y_train[i], alpha=0.1)
```

حلقه بیرونی همان epoch های ما هستند. به ازای هر ایپاک، تمام داده های موجود در دیتاست یک بار دیده میشود. حلقه داخلی هم برای بررسی 4 داده موجود است. چون تعداد داده ها کم بود برای راحتی محاسبه گرادیان ها، از 4 batch تایی استفاده نکردم.

پس از 500 ایپاک، پیشبینی جدید مدل برای نمونه ها به این صورت است:

```
print('After Learning')
print(forward(x_train[0], W1, W2, b1, b2))   # x = 0, 0    y = 1
print(forward(x_train[1], W1, W2, b1, b2))   # x = 0, 0    y = 0
print(forward(x_train[2], W1, W2, b1, b2))   # x = 0, 0    y = 0
print(forward(x_train[3], W1, W2, b1, b2))   # x = 0, 0    y = 1
```

```
After Learning
[[0.95340432]]
[[0.0101688]]
[[0.01041628]]
[[0.99339687]]
```

میبنییم که آموزش مدل به خوبی صورت گرفته و مقادیر احتمال پیشبینی شده توسط مدل به واقعیت بسیار نزدیک هستند. اگر با تنظیم یک threshold، مقادیر احتمال را به 0 یا 1 تبدیل کنیم، میبینیم که دقت مدل برای این 4 نمونه به 100 درصد رسیده اما برای درک بهتر مدل، خود مقادیر احتمال را چاپ کرده ام.

لازم به ذکر است که روش استفاده شده برای train این مدل، SGD است. از کتابخانه ای غیر از numpy هم استفاده نشده. کد این سوال هم درفایل MLP_with_numpy در فایل زیپ موجود است.

Q6)

برای حل این مسئله، میتوانیم هم از مدل های کانولوشنی استفاده کنیم هم مدل های MLP. با مدل های MLP ساده تر آغاز میکنیم و در صورتی که به دقت مطلوب نرسیدیم، سراغ مدل های کانولوشنی میرویم.

برای شروع، از دو لایه مخفی استفاده میکنیم. لایه ورودی 28*28 است و لایه خروجی 10 نورون دارد (زیرا مسئله دسته بندی ده کلاسه است). بنابراین تعداد نورون های دو لایه وسط را طوری انتخاب میکنیم که لایه به لایه کم تر شود. برای مثلا درلایه دوم 256 نورون و در لایه سوم، 128 نورون قرار میدهیم.

ابتدا کتابخانه را لود کرده و دو لودر برای داده های تست و ترین قرار میدهیم:

```
[40] train =  MNIST('/files/', train=True, download=True, transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]))
     test =  MNIST('/files/', train=False, download=True, transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]))

     train_loader = torch.utils.data.DataLoader(train, batch_size=64, shuffle=True)
     test_loader = torch.utils.data.DataLoader(test, batch_size=64, shuffle=True)
```

مدل را طبق توضیح بالا ایجاد میکنیم:

```python
class MLP(nn.Module):
  '''
    Multilayer Perceptron.
  '''
  def __init__(self):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Flatten(),
      nn.Linear(28 * 28, 256),
      nn.ReLU(),
      nn.Linear(256, 128),
      nn.ReLU(),
      nn.Linear(128, 10),
    )


  def forward(self, x):
    '''Forward pass'''
    return self.layers(x)
```

سپس توابع لاس و اپتیمایزر را مشخص میکنیم:

```python
[44] loss_function = nn.CrossEntropyLoss()
     optimizer = torch.optim.Adam(mlp.parameters(), lr=1e-2)
     n_epochs = 10
     val_per_epoch = 1
```

همچنین تعداد ایپاک ها و ریت انجام ولیدیشن.

این تابع برای اندازه گیری دقت مدل به کار میرود:

```python
[45] def accuracy(yhat, y):
       pred_idx = yhat.max(1, keepdim=True)[1]
       correct = pred_idx.eq(y.view_as(pred_idx)).sum().item()
       return correct / len(y)
```

از یک دیکشنری برای ذخیره هیستوری مدل استفاده میکنیم:

```
history = dict()
history['train_loss'] = list()
history['train_acc'] = list()
history['val_loss'] = list()
history['val_acc'] = list()
```

سپس فرایند ترین را با توجه به سینتکس پایتورچ شروع میکنیم:

```
for epoch in range(n_epochs):
  running_loss = 0.0
  running_acc = 0
  mlp.train()

  for idx, (x, y) in enumerate(tqdm(train_loader)):
    optimizer.zero_grad()
    yhat = mlp(x)
    loss = loss_function(yhat, y)
    loss.backward()
    optimizer.step()

    running_loss += loss.item()
    running_acc += accuracy(yhat, y)

  running_loss /= len(train_loader)
  running_acc /= len(train_loader)
  history['train_loss'].append(running_loss)
  history['train_acc'].append(running_acc)
  print(f"epoch = {epoch}\ttraining loss = {running_loss}\ttrainig accuracy = {running_acc}")
```

و در ایپاک های مشخص(در این جا هر 1 ایپاک) فرایند ولیدیشن را انجام میدهیم:

```
if epoch % val_per_epoch == val_per_epoch - 1:
  running_loss = 0.0
  running_acc = 0
  mlp.train()
  with torch.no_grad():
    for idx, (x, y) in enumerate(tqdm(test_loader)):
      yhat = mlp(x)
      loss = loss_function(yhat, y)

      running_loss += loss.item()
      running_acc += accuracy(yhat, y)

    running_loss /= len(test_loader)
    running_acc /= len(test_loader)
    history['val_loss'].append(running_loss)
    history['val_acc'].append(running_acc)
    print(f"epoch = {epoch}\tvalidation loss = {running_loss}\tvalidation accuracy = {running_acc}")
```

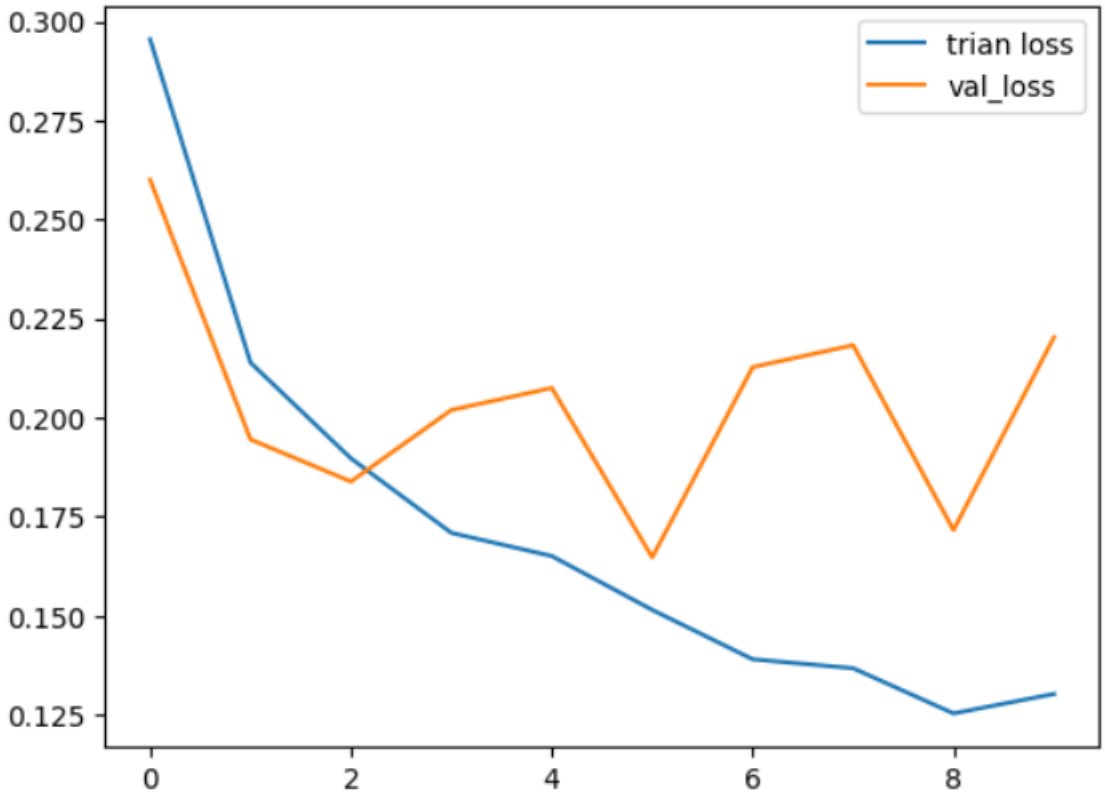
می بینیم که مدل به دقت مورد نظر یعنی بالای 95 رسیده:

```
epoch = 7       training loss = 0.13670013420894914     trainig accuracy = 0.9672174840085288
100%|██████████| 157/157 [00:03<00:00, 46.05it/s]
epoch = 7       validation loss = 0.2182631044393512    validation accuracy = 0.9539211783439491
100%|██████████| 938/938 [00:22<00:00, 41.19it/s]
epoch = 8       training loss = 0.12528903029825805     trainig accuracy = 0.9700659648187633
100%|██████████| 157/157 [00:03<00:00, 47.55it/s]
epoch = 8       validation loss = 0.17160165696671814   validation accuracy = 0.9633757961783439
```

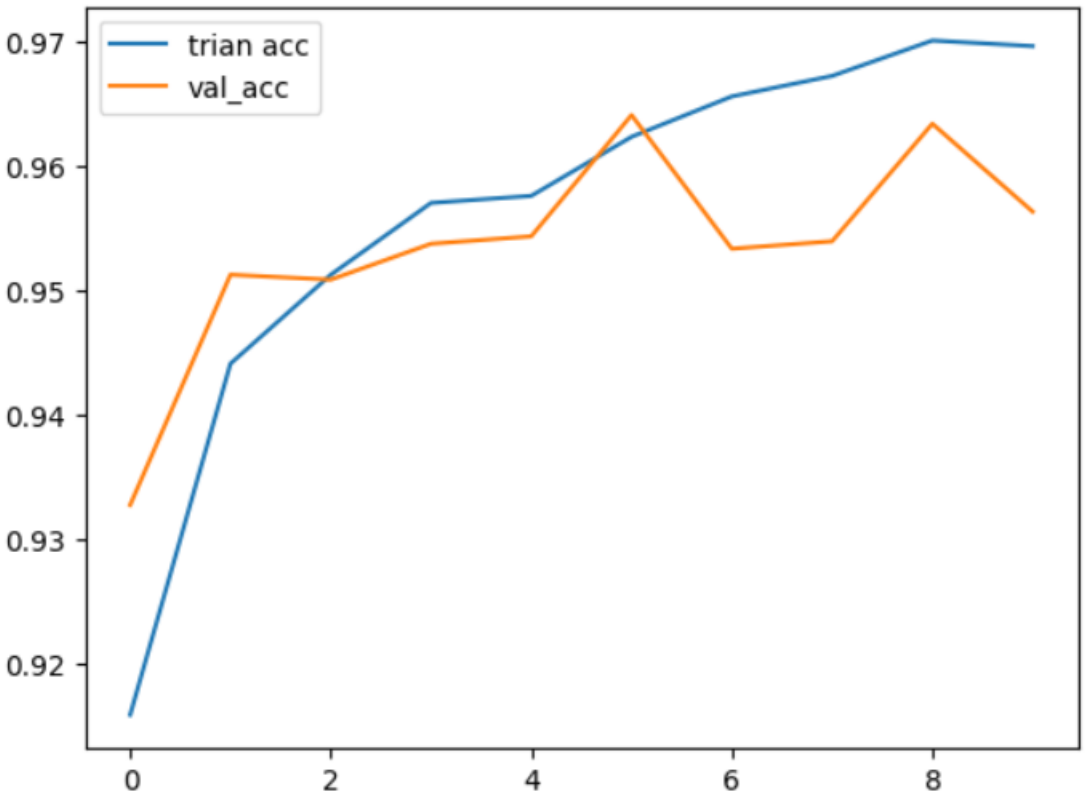
البته در ادامه این مدل کمی overfit شد و دقت از 96 مجددا به 95 رسید اما در نهایت پس از 10 ایپاک همچنان بالای 95 بود.

حالا با استفاده از هیستوری ذخیره شده، نمودار لاس و دقت را رسم میکنیم.

نمودار loss:



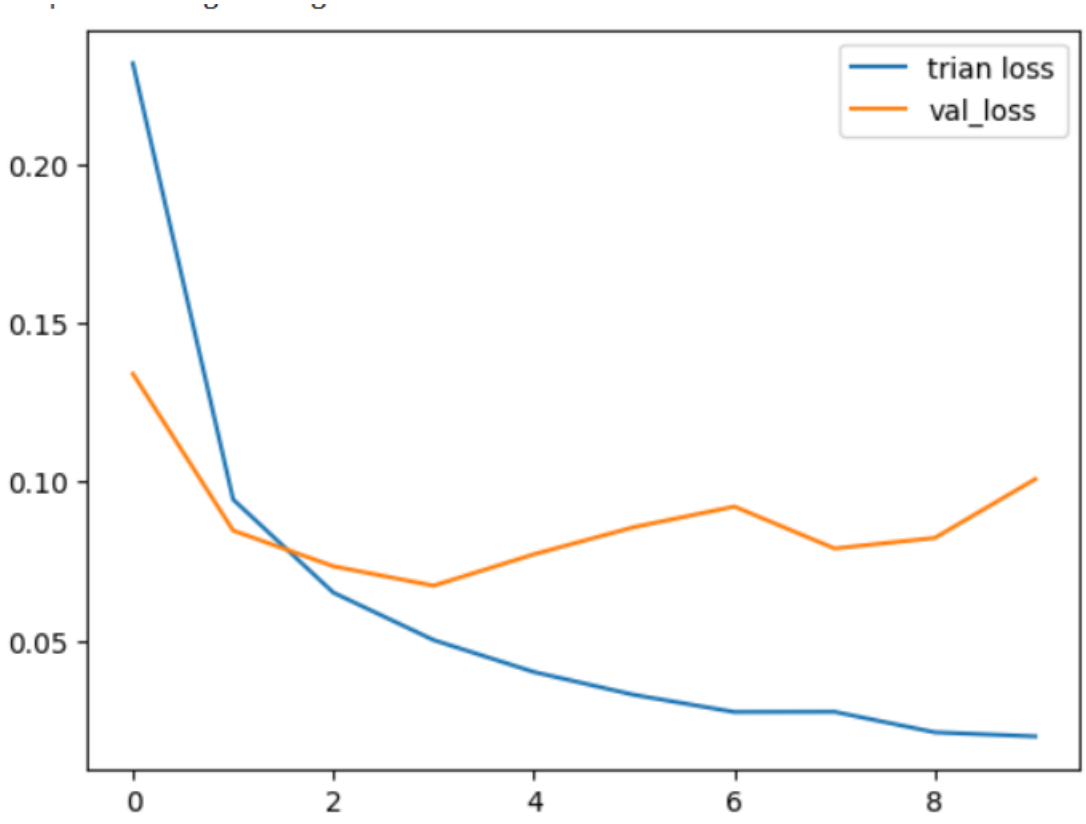
نمودار accuracy:



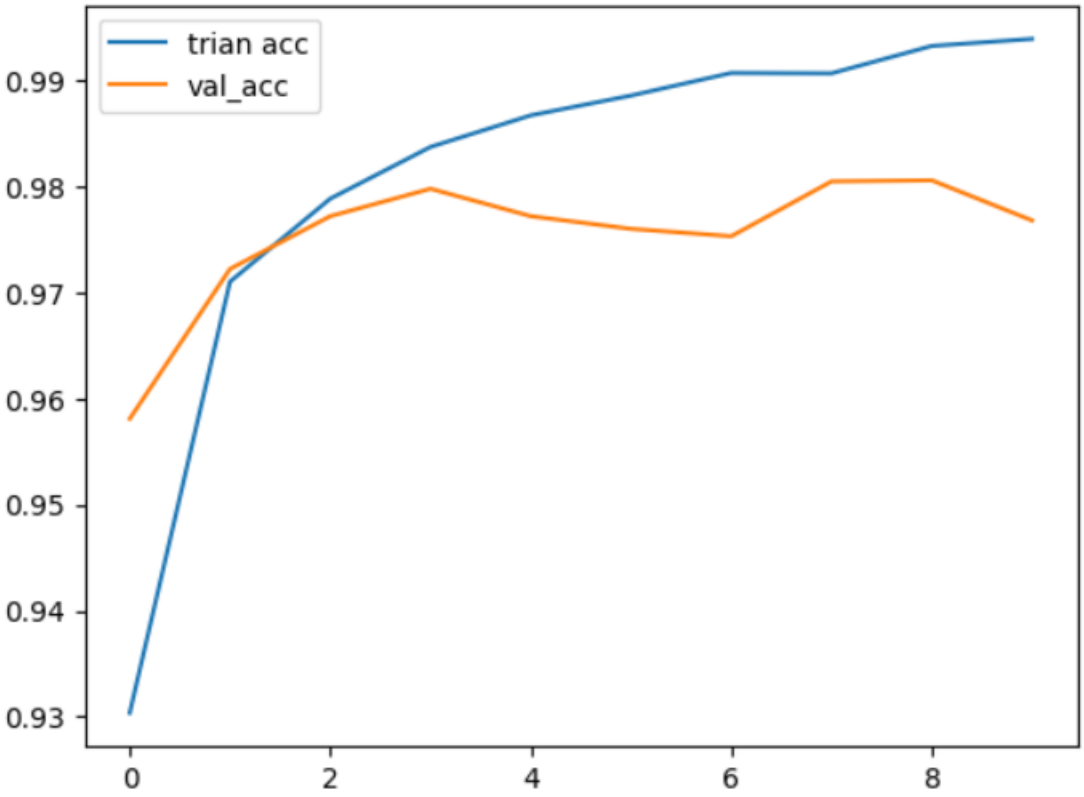
با توجه به نمودارها میتوان متوجه شد که مدل، پس از ایپاک 5 دچار overfitting شده.

این بار لرنینگ ریت را از 0.01 به 0.001 تغییر میدهیم و نتایج را چک میکنیم:

نمودار loss:



نمودار accuracy:



باز هم مدل دچار اورفیتینگ شده چون ساختار مدل تغییر نکرده و داده ها هم بیشتر نشده. اما میبینیم که عملکرد مدل در کل بهبود داشته و دقت train و test یک تا دو درصد بهتر شده. همچنین میبنییم که نوسان زیاد مدل که احتمالا ناشی از لرنینگ ریت زیاد بود کاهش یافته.

کد این سوال در فایل MNIST_MLP_model در فایل زیپ موجود است.