آتوسا مالمیر چگینی

97106251

گزارش پروژه ی شبکه عصبی

*** در این پروژه 7 بخش مختلف وجود داشتند که در این جا ابتدا کد هر کدام را توضیح داده و سپس نتایج به دست آمده به ازای پارامتر های مختلف را بررسی میکنیم .

***GUI های این پروژه بسیار ساده هستند چون هدف از این پروژه GUI نبود به همین دلیل من وقتی برای آن ها نگذاشتم .

بخش اول :

هدف :

در این قسمت از پروژه خواسته شده بود که تعدادی تابع ریاضیاتی خاص در نظر بگیریم و نقاطی روی این توابع ایجاد کنیم (هر نقطه در واقع یک زوج مرتب (x, y) است .) . و این نقاط را به عنوان نقاط آموزشی به یک شبکه عصبی چند لایه بدهیم تا آن ها را یاد بگیرد و با استفاده از این نقاط تابعی به دست بیاورد. با رسم این تابع به دست آمده توسط MLP در کنار تابع اصلی میتوان متوجه شد که شبکه ی عصبی تا چه میزان توانسته است تابع را درست به دست بیاورد.

توضیح کد :

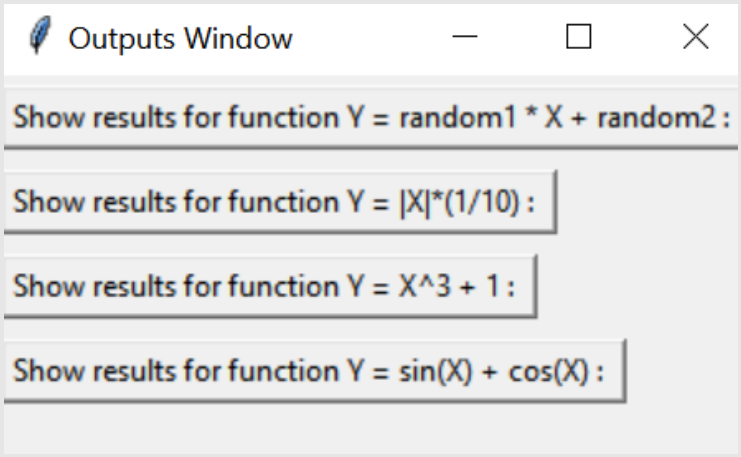
برای انجام این بخش ما 4 تابع زیر را برای آموزش MLP انتخاب کردیم :

1) Y = random1 * X + random2
2) Y = |X|*(1/10)
3) Y = X^3 + 1
4) Sin(X) + cos(X)

به تعداد number_of_points که در ابتدای کد مشخص شده است نقطه از محور X ها انتخاب میکنیم و این نقاط را به عنوان ورودی به هر کدام از 4 تابع بالا میدهیم و در نهایت به تعداد number_of_points نقطه ی آموزشی داریم .

حال 4 تا MLPRegressor برای 4 تابع بالا ایجاد میکنیم و پارامتر های مختلف از قبیل آلفا ، تعداد و سایز لایه های پنهان ، تعداد چرخه ها ، نوع تابع activation و موارد دیگر را مشخص میکنیم .

در ادامه یک GUI بسیار ساده برای این بخش ایجاد کرده ایم . که به شکل زیر است :



که با کلیک کردن هر کدام شبکه عصبی شروع به یاد گرفتن آن تابع میکند و بعد از این که همه را یادگرفت با بستن این پنجره تابع اصلی(آبی) و تابع پیشبینی شده(قرمز) رسم میشود.

سپس با استفاده از دستور clf.predict(X) برای هر تابع خروجی های پیشبینی شده به ازای x ها را به دست می آوریم.

در آخر با plot کردن تابع اصلی و تابع پیشبینی شده برای هر کدام از 4 تابع بالا نتایج آموزش شبکه عصبی را نشان میدهیم .

بررسی تاثیر پارامتر های مختلف :

تعداد نقاط ورودی (number_of_points) : مشخصا هر چقدر تعداد نقاط آموزشی بیشتر باشد نتیجه با دقت بالاتری به دست خواهد آمد.

Number_of_points = 500

Function 1:

```
Iteration 2956, loss = 0.09069373
Iteration 2957, loss = 0.09063122
Iteration 2958, loss = 0.09054578
Iteration 2959, loss = 0.09047308
Iteration 2960, loss = 0.09039404
Iteration 2961, loss = 0.09031735
Iteration 2962, loss = 0.09026217
Iteration 2963, loss = 0.09016245
Training loss did not improve more than tol=0.000100 for 10 consecut
```

Function2 :

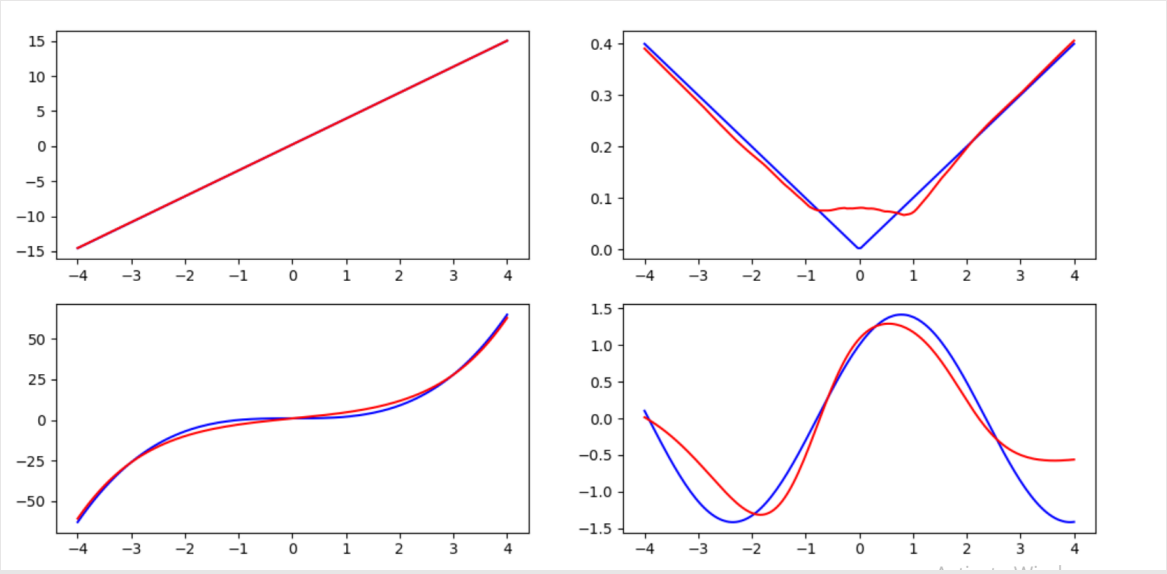
```
Iteration 34, loss = 0.00178809
Iteration 35, loss = 0.00173229
Iteration 36, loss = 0.00168626
Iteration 37, loss = 0.00164877
Iteration 38, loss = 0.00160837
Iteration 39, loss = 0.00157550
Iteration 40, loss = 0.00154445
Training loss did not improve more than tol=0.000100 for 10 consecut
```

Function3 :

```
Iteration 58284, loss = 208.92776185
Iteration 58285, loss = 208.95473031
Iteration 58286, loss = 208.90961193
Iteration 58287, loss = 208.99152872
Iteration 58288, loss = 208.94621743
Training loss did not improve more than tol=0.000100 for 10 consecut
```


Function4 :

```
Iteration 865, loss = 0.36079929
Iteration 866, loss = 0.36070663
Iteration 867, loss = 0.36077747
Iteration 868, loss = 0.36055247
Iteration 869, loss = 0.36058553
Training loss did not improve more than tol=0.000100 for 10 consecut
```



حال تعداد نقاط آموزشی را 2000 تا در نظر میگیریم .

Function1 :

```
Iteration 2676, loss = 0.09805776
Iteration 2677, loss = 0.09796309
Iteration 2678, loss = 0.09790248
Iteration 2679, loss = 0.09787904
Iteration 2680, loss = 0.09786275
Training loss did not improve more than tol=0.000100 for 10 consecut
```

Function2 :

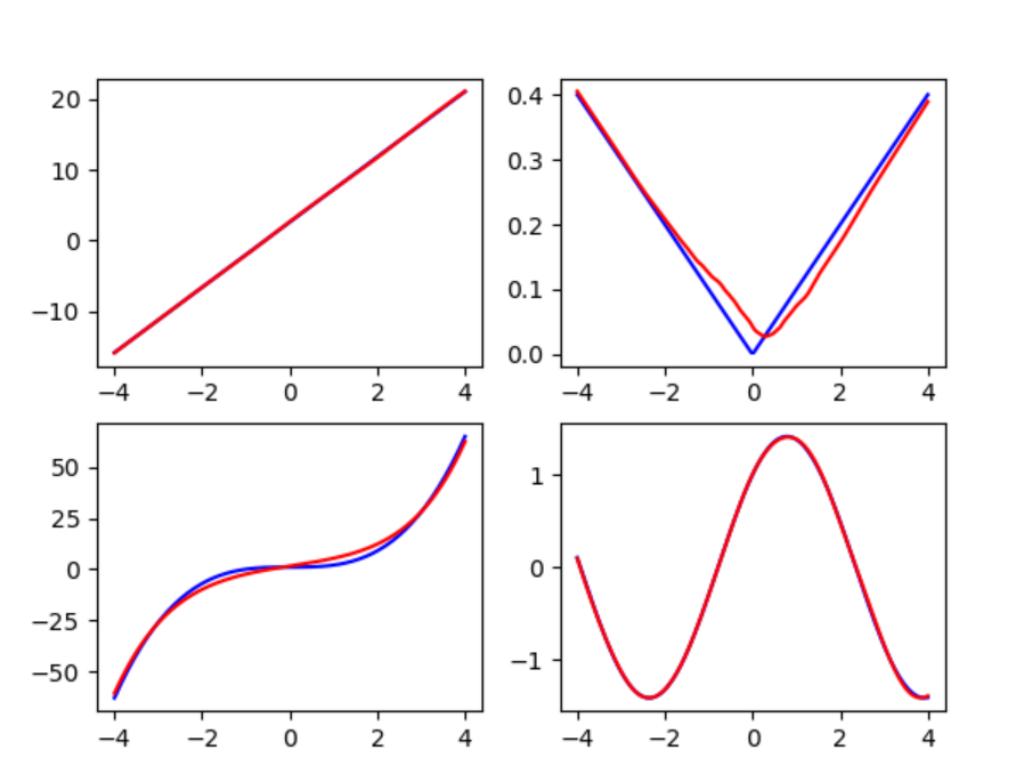
```
Iteration 24, loss = 0.00149020
Iteration 25, loss = 0.00144986
Iteration 26, loss = 0.00141383
Iteration 27, loss = 0.00138149
Iteration 28, loss = 0.00135189
Training loss did not improve more than tol=0.000100 for 10 consecut
```

Function3 :

```
Iteration 36051, loss = 163.60820284
Iteration 36052, loss = 163.56420249
Iteration 36053, loss = 163.59416529
Iteration 36054, loss = 163.60259267
Iteration 36055, loss = 163.53010389
Iteration 36056, loss = 163.63211192
Training loss did not improve more than tol=0.000100 for 10 consecut
```

Function4 :

```
Iteration 2565, loss = 0.13290813
Iteration 2566, loss = 0.13257211
Iteration 2567, loss = 0.13233399
Iteration 2568, loss = 0.13228557
Iteration 2569, loss = 0.13233733
Iteration 2570, loss = 0.13243551
Training loss did not improve more than tol=0.000100 for 10 consecut
```



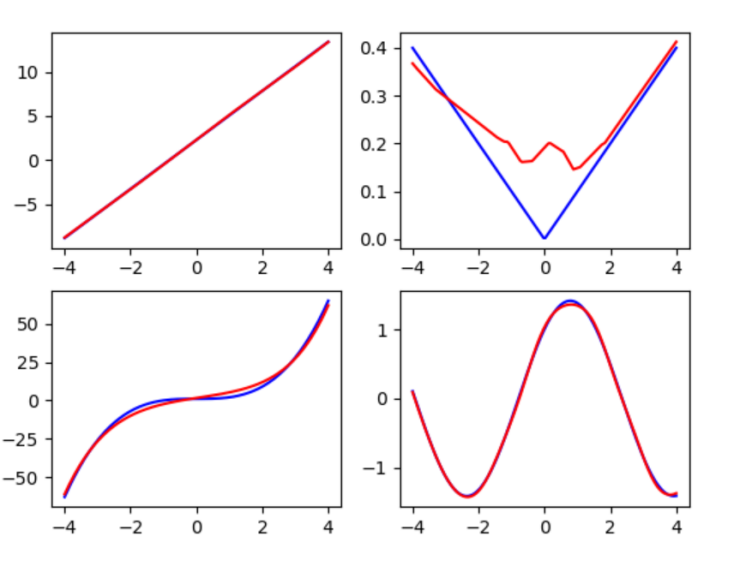
*** همان طور که معلوم است با زیاد کردن تعداد نقاط آموزشی تابع پیشبینی شده به تابع اصلی نزدیک تر میشود .

میزان پیچیدگی تابع : همانطور که از نتایج بالا معلوم است معمولا توابع ساده مثل توابع خطی حتی به ازای تعداد نقاط آموزشی کم نتیجه ب بعتری میدهد و حدس زدن آن برای شبکه عصبی ساده تر است.

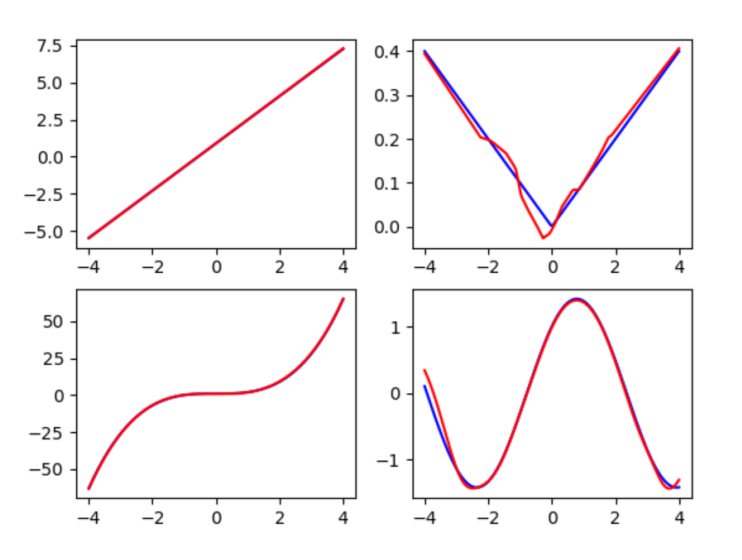
اما توابعی که قله یا دره ی تیز دارند مثل تابع شماره ی 2 ، معمولا پیشبینی کردنشان برای شبکه عصبی سخت تر است .

تعداد لایه های شبکه عصبی و تعداد نورون های هر لایه : تنظیم این مقادیر توسط پارامتری به نام hidden_layer_size در MLPRegressor انجام میشود .

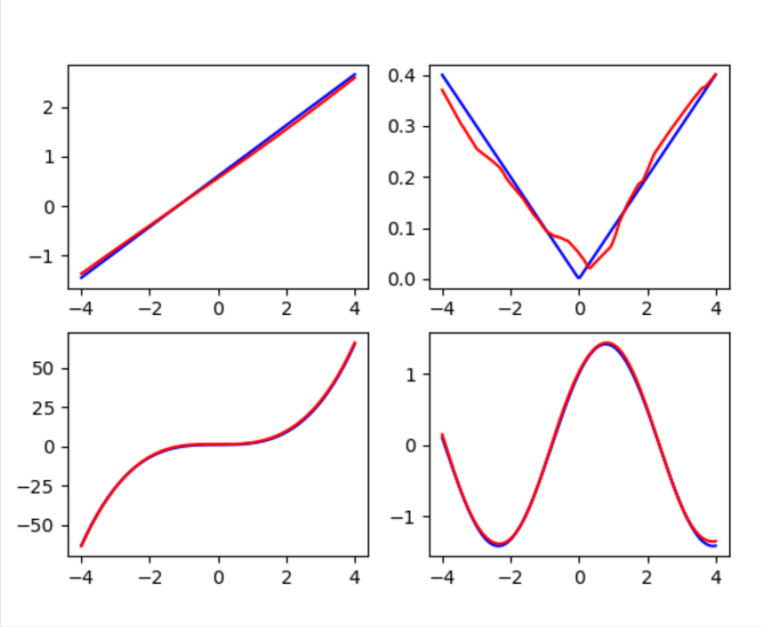
Hidden_layer_size = (20,) = 1 hidden layer with 20 neurons



Hidden_layer_size = (20, 20) = 2 hidden layers with 20 neurons in each



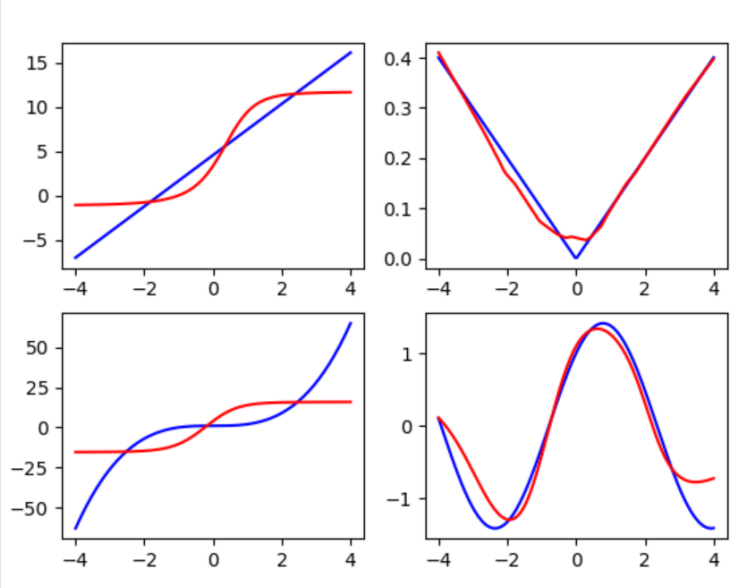
Hidden_layer_size = (20, 20, 20) = 3 hidden layers with 20 neurons in each



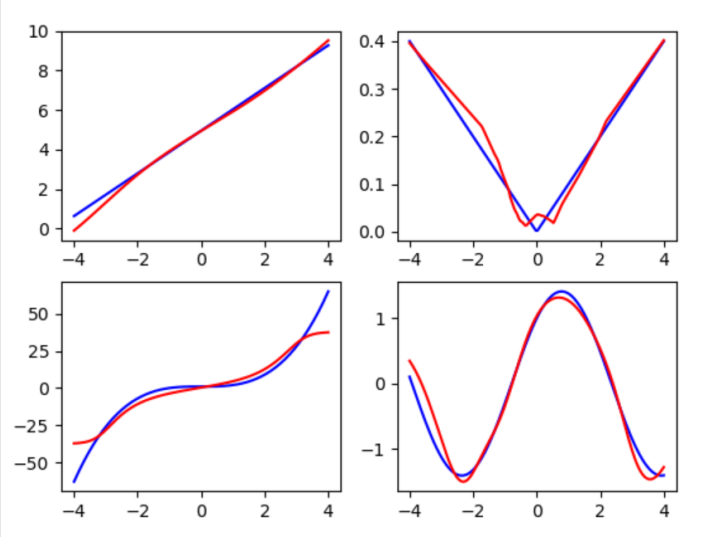
همان طور که مشخص است زیاد کردن تعداد لایه ها و نورون ها تا حدی باعث بهتر شدن نتیجه میشود ولی از یک حدی به بعد تغییر زیادی در نتیجه ی ایجاد شده و در دقت به دست آمده ندارد.

تعداد چرخه ها : تا این جای کار تعداد چرخه ها را بسیار زیاد گرفتیم تا بهترین نتیجه به دست بیاید.

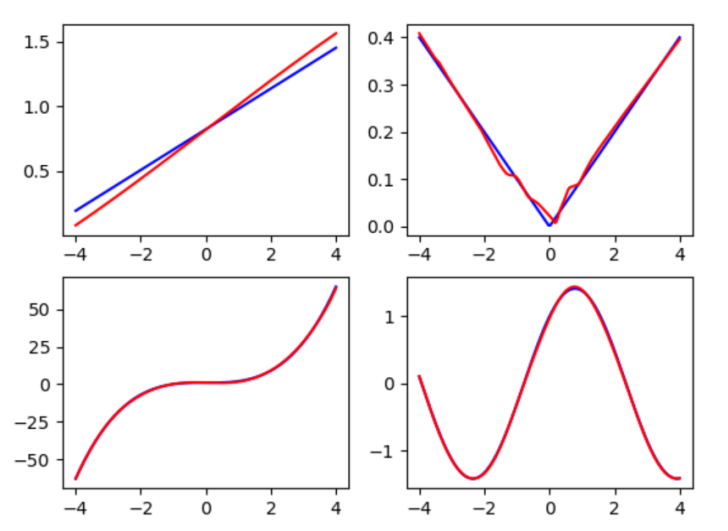
Max_iteration = 100



Max_iteration = 300



Max_iteration = 1000

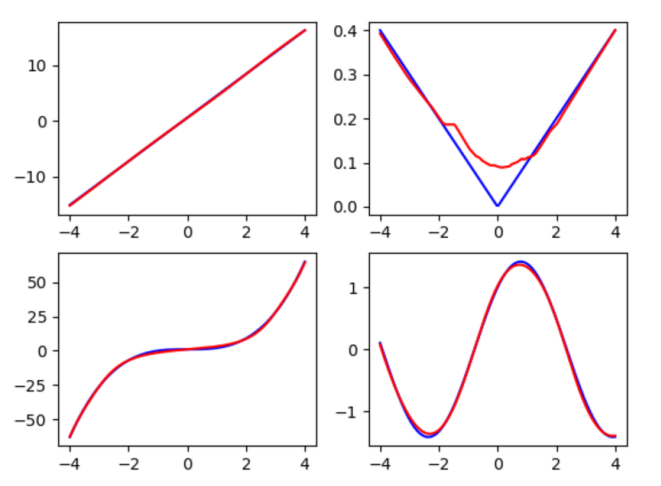


***در واقع میتوان گفت هر چقدر تعداد چرخه ها بیشتر باشد شبکه عصبی دقت بیشتری خواهد داشت.

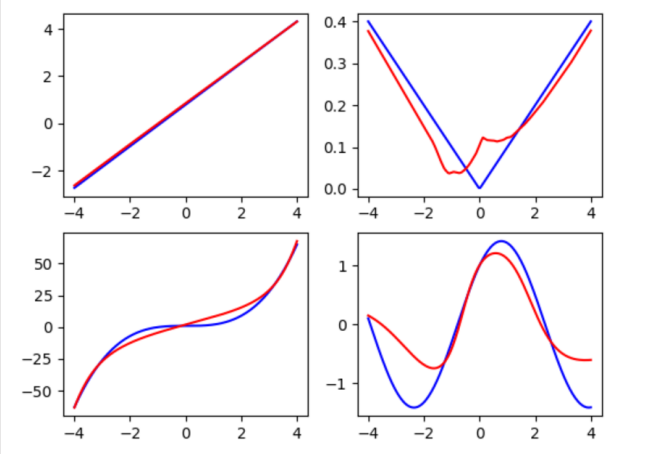
وسعت دامنه ی ورودی :این پارامتر را در بخش پایین تغییر میدهیم.

```
x1 = np.random.uniform(-15, 15, size=number_of_points)
```

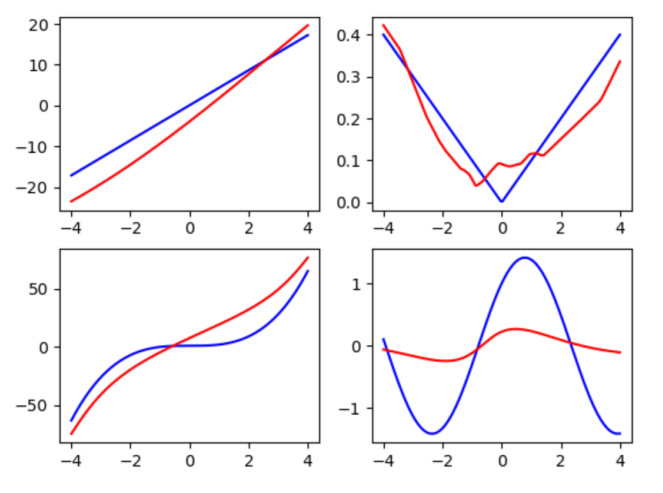
(-10, 10) :



(-50, 50) :



(-150, 150) :



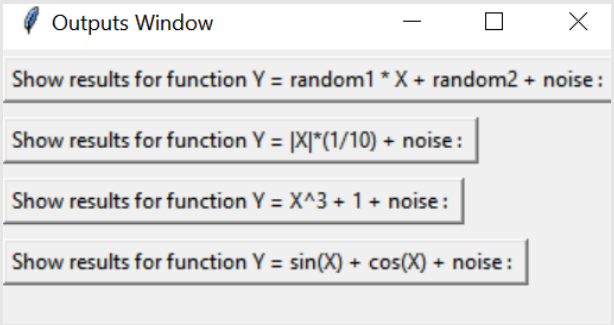
همانطور که از نمودار توابع مشخص است هر چقدر وسعت دامنه ی ورودی بیشتر باشد دقت شبکه کم تر میشود چون نقاط انتخاب شده از هم فاصله ی بیشتر ی خواهند داشت و شبکه عصبی اطلاعات را راجع به بخشی از تابع ندارد و بنابراین آن قسمت را به خوبی یاد نمیگیرد.

بخش دوم :

هدف :

در این بخش باید به توابع بخش 1 مقداری noise اضافه کنیم و بعد به شبکه عصبی ورودی بدهیم و نتیجه ی آموزش داده های دارای noise را بررسی کنیم .

GUI این بخش :



توضیح کد :

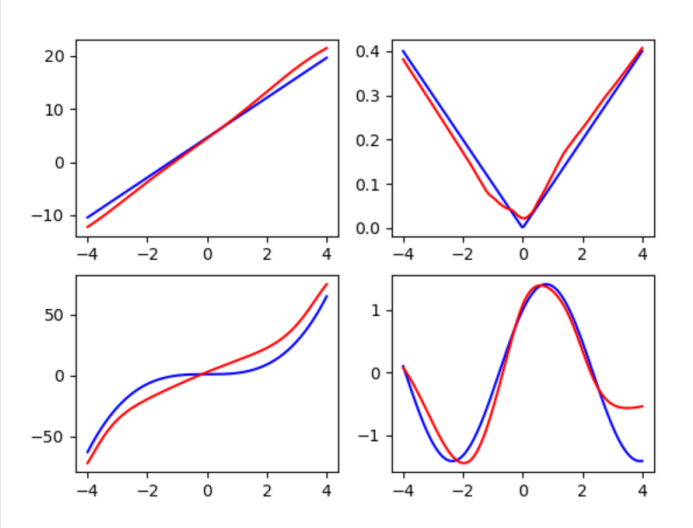
کد این بخش دقیقا مانند بخش اول است با این تفاوت که در این جا به هر تابع مقداری noise اضافه کردیم :

```
np.random.seed(3)
noise = np.random.randn(number_of_points, )
```

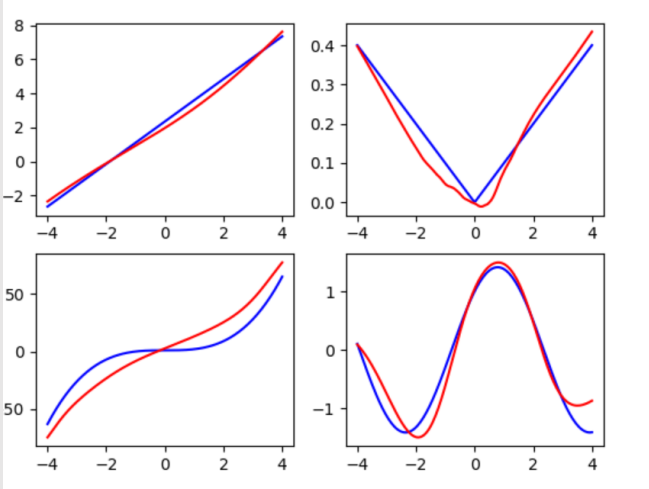
دو خط بالا در واقع به این معنی است که یک آرایه ساختیم به تعداد نقاط ورودی خانه دارد و در هر خانه عددی رندم بین 3 ,3- داریم که این مقدار noise را به هر کدام از نقاط ورودی اضافه میکنیم .

بررسی تاثیر مقدار noise ها مختلف :

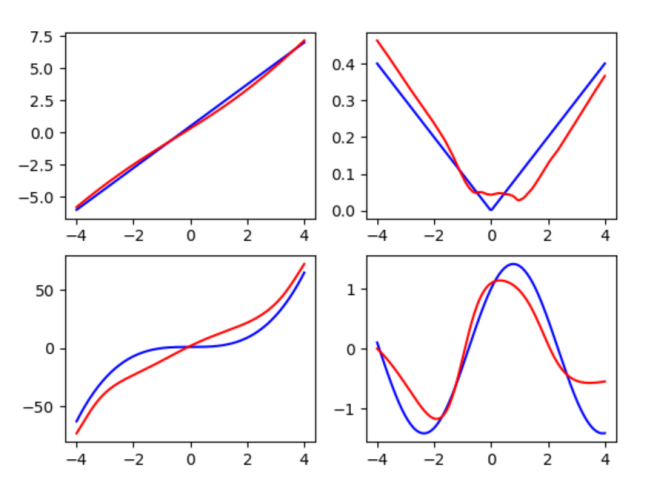
1) Np.random.seed(3) :



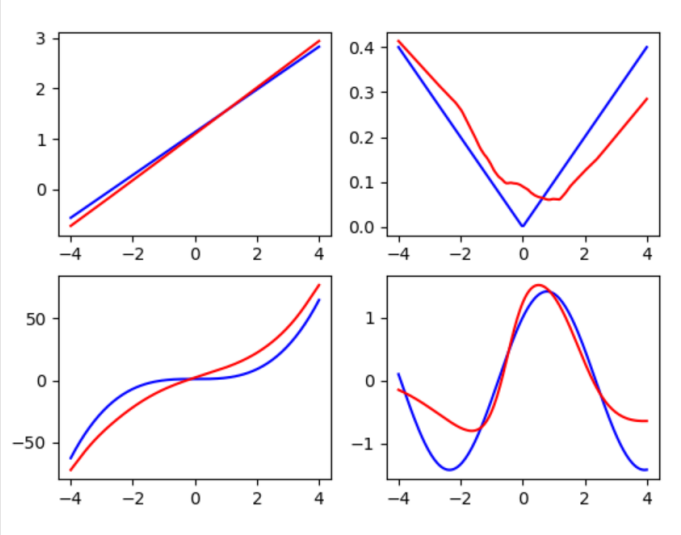
2) Np.random.seed(10) :



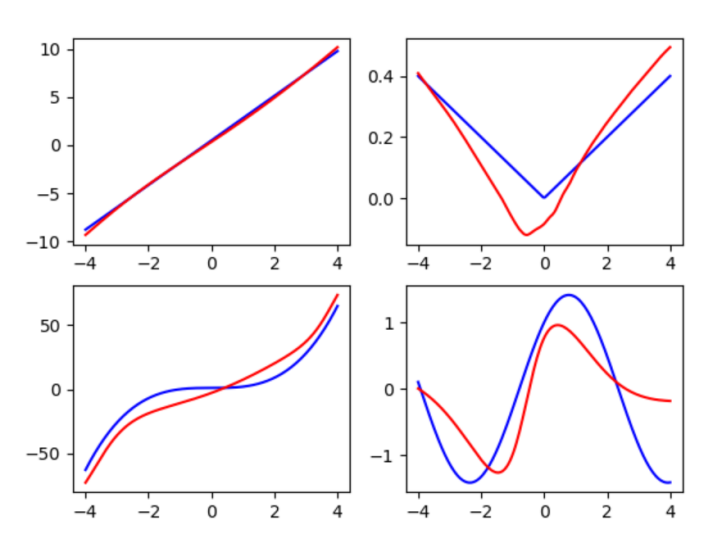
3) Np.random.seed(20) :



4) Np.random.seed(100) :



5) Np.random.seed(1000) :



همان طور که مشخص است هر چقدر noise اضافه شده به داده های ورودی بیشتر باشد حدس زدن تابع اولیه برای شبکه عصبی سخت تر خواهد بود .

در بخش اول دیدیم که با تغییر پارامتر های مختلف میتوان قدرت و دقت شبکه عصبی را برای یاد گرفتن تابع افزایش داد . در بخش دوم تغییر دادن پارامتر ها از قبیل تعداد داده های ورودی و یا تعداد لایه ها و نورون ها شاید بتواند عملکرد شبکه را بهتر کند اما اگر randomness اضافه شده به توابع خیلی زیاد باشد ممکن است تلاش پارامتر ها ☺ برا ی بهتر کردن یادگیری چندان تاثیر گذار نباشد .

بخش سوم :

هدف :

هدف از این بخش دقیقا مانند بخش اول بررسی دقت شبکه عصبی در یادگیری توابع به ازای پارامتر های مختلف است . با این تفاوت که توابع این بخش ابعاد بیشتری دارند و منطقا پیچیده تر هستند .

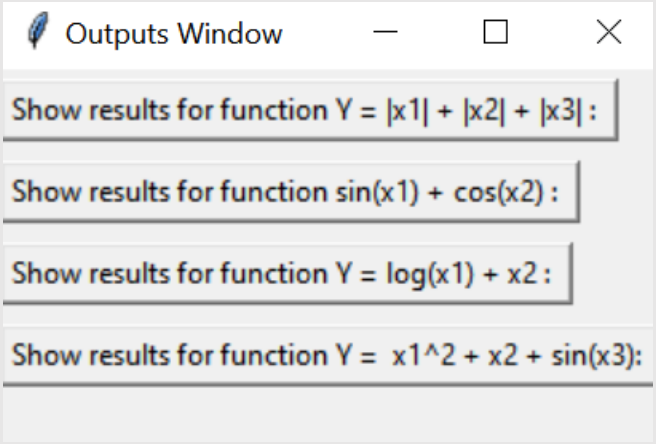
توابع در نظر گرفته شده برای این بخش عبارت اند از :

1) Y = |x1| + |x2| + |x3|
2) Y = sin(x1) + cos(x2)
3) Y = log(x1) + x2
4) Y = x1^2 + x2 + sin(x3)

تابع اول و چهارم 4 بعدی و تابع دوم و سوم 3 بعدی هستند .

***در ضمن در این بخش به دلیل زیاد بودن ابعاد توابع ان ها را رسم نمیکنیم و تنها با مقادیر عددی میزان یادگیری را بررسی میکنیم . از مقداری به نام loss استفاده میکنیم که هر چقدر کم تر باشد به این معنی است که شبکه عصبی بهتر و دقیق تر تابع ورودی داده شده را یاد گرفته است .

GUI این بخش :



توضیح کد :

کد این بخش نیز بسیار شبیه کد بخش اول است با این تفاوت که چون ابعاد توابع بیشتر هستند نقاط رندم را تنها برای یک X ایجاد نکردیم بلکه برای هر سه x1, x2, x3 این نقاط رندم را به دست آوردیم. از آنجا که بعضی از توابع X دو بعدی و بعضی X سه بعدی دارند با استفاده از دو خط زیر این دو نوع X ورودی را با reshape کردن x1, x2, x3 به صورت زوج دو تایی یا سه تایی در می آوریم .

```
X1 = np.reshape((x1, x2), [number_of_points, 2])
X2 = np.reshape((x1, x2, x3), [number_of_points, 3])
```

بقیه بخش های کد دقیقا مانند بخش اول است .

بررسی تاثیر پارامتر های مختلف :

گفتیم هدف این بخش دقیقا مانند بخش اول است و تنها توابع پیچیده تر شده اند .به این جهت تاثیر پارامتر های مختلف دقیقا مانند بخش اول است یعنی :

1) با زیاد شده تعداد نقاط ورودی مقدار loss کم تر شده پس دقت بیشتر میشود.

Function1 :

Number_of_points = 300 :

Iteration 995, loss = 143.23878229
Iteration 996, loss = 143.16034940
Iteration 997, loss = 143.08060111
Iteration 998, loss = 143.00048662
Iteration 999, loss = 142.92066775
Iteration 1000, loss = 142.84273625

Number_of_points = 1000 :

Iteration 996, loss = 47.07808501
Iteration 997, loss = 47.01020402
Iteration 998, loss = 46.94019210
Iteration 999, loss = 46.87325090
Iteration 1000, loss = 46.80393198

Function2 :

Number_of_points = 300 :

Iteration 46, loss = 0.49631217
Iteration 47, loss = 0.49560594
Iteration 48, loss = 0.49317605
Iteration 49, loss = 0.49470983
Iteration 50, loss = 0.49515169

Number_of_points = 1000 :

Iteration 46, loss = 0.48097631
Iteration 47, loss = 0.48069106
Iteration 48, loss = 0.48048620

Iteration 49, loss = 0.48042081
Iteration 50, loss = 0.48027248

Function3 :
Number_of_points = 300 :
Iteration 196, loss = 36.89524443
Iteration 197, loss = 36.89247594
Iteration 198, loss = 36.92419032
Iteration 199, loss = 36.92279377
Iteration 200, loss = 36.88899333

Number_of_points = 1000 :
Iteration 196, loss = 35.73419170
Iteration 197, loss = 35.71908994
Iteration 198, loss = 35.70740486
Iteration 199, loss = 35.70195101
Iteration 200, loss = 35.69930805

Function4 :
Number_of_points = 300 :
Iteration 196, loss = 4710.25876952
Iteration 197, loss = 4709.82995872
Iteration 198, loss = 4709.39367378
Iteration 199, loss = 4708.96454860
Iteration 200, loss = 4708.54252829
Number_of_points = 1000 :
Iteration 195, loss = 4596.39384632
Iteration 196, loss = 4595.39101967
Iteration 197, loss = 4594.37386271
Iteration 198, loss = 4593.33307109
Iteration 199, loss = 4592.33215958

2) میزان پیچیدگی تابع : همانطور که از نتایج بالا معلوم است معمولا توابع ساده حتی به ازای تعداد نقاط آموزشی کم نتیجه بهتری میدهد و حدس زدن آن برای شبکه عصبی ساده تر است (مثل تابع

اول که مقدار loss ان کم تر است .) اما تابع چهارم که 4 بعدی است و تابع سختی محسوب میشود مقدار loss بیشتری داشته است.

3) تعداد لایه های شبکه و تعداد نورون های هر لایه : زیاد کزدن تعداد نورون ها و لایه ها تا حدی باعث میشود که شبکه عصبی دقیق تر شود اما از یک جایی به بعد تاثیری بر دقت شبکه ندارد.

برای بررسی این گفته تابع سوم را در نظر میگیریم :

Function3 :

Hidden_layer_sizes = (20,) = 1 hidden layer with 20 neurons

Iteration 495, loss = 37.91031771

Iteration 496, loss = 37.97500151

Iteration 497, loss = 37.88628366

Iteration 498, loss = 37.92047289

Iteration 499, loss = 37.94417037

Hidden_layer_sizes = (20, 20) = 2 hidden layers with 20 neurons in each

Iteration 496, loss = 37.47071533

Iteration 497, loss = 37.46447022

Iteration 498, loss = 37.46596299

Iteration 499, loss = 37.46698249

Iteration 500, loss = 37.46578104

Hidden_layer_sizes = (20, 20, 20) = 3 hidden layers with 20 neurons in each

Iteration 496, loss = 35.71066390

Iteration 497, loss = 35.70854074

Iteration 498, loss = 35.71357008

Iteration 499, loss = 35.69765970

Iteration 500, loss = 35.71995312

4) تعداد چرخه ها : هر چه تعداد چرخه ها بیشتر باشد شبکه عصبی بهتر تابع را یاد میگیرد. برای بررسی این ویژگی تابع 4 را انتخاب میکنیم .

Function4 :

Max_iteration = 300
Iteration 296, loss = 4495.66225793
Iteration 297, loss = 4494.62926624
Iteration 298, loss = 4493.59374677
Iteration 299, loss = 4492.58548736
Iteration 300, loss = 4491.54955471

Max_iteration = 1000
Iteration 996, loss = 3739.51630949
Iteration 997, loss = 3738.74272860
Iteration 998, loss = 3737.92296960
Iteration 999, loss = 3737.16797492
Iteration 1000, loss = 3736.35926815

Max_iteration = 5000
Iteration 4996, loss = 2257.85952860
Iteration 4997, loss = 2257.81288890
Iteration 4998, loss = 2257.77802115
Iteration 4999, loss = 2257.74238539
Iteration 5000, loss = 2257.70047321

وقتی تعداد چرخه ها بیشتر شد ، مقدار loss کم تر شد.

5) وسعت دامنه ورودی : معمولا افزایش دامنه ورودی باعث کم تر شدن دفت MLP میشود به این دلیل که در بعضی از قسمت عای تابع نقطه ی آموزشی به شبکه داده نشده است .

برای بررسی این ویژگی تابع چهارم را در نظر میگیریم :

Function4 :

x1, x2, x3 = (-10, 10)
Iteration 996, loss = 641.14254435
Iteration 997, loss = 640.91167979
Iteration 998, loss = 640.69066417
Iteration 999, loss = 640.46902344
Iteration 1000, loss = 640.24838225

x1, x2, x3 = (-20, 20)
Iteration 996, loss = 13110.41700475
Iteration 997, loss = 13108.82602310
Iteration 998, loss = 13107.18341712
Iteration 999, loss = 13105.56308104
Iteration 1000, loss = 13103.96717750

x1, x2, x3 = (-50, 50)
Iteration 996, loss = 616706.35268941
Iteration 997, loss = 616694.03214441
Iteration 998, loss = 616681.49713999
Iteration 999, loss = 616669.06195825
Iteration 1000, loss = 616656.62572392

میبینیم که با زیاد شدن دامنه ی ورودی مقدار loss به شدت افزایش میابد و دقت به شدت کاهش میابد .

***یکی دیگر از پارامتر هایی که در زمان کار کردن با شبکه ی عصبی بسیار مهم به نظرم آمد ، نوع تابع فعال سازی (activation) است .که در sklearn 3 حالت logistic و relu و tanh را دارد .

به نظر در توابع پیچیده انتخاب logistic اصلا کار خوبی نیست و دقت را پایین می آورد ؛ اما در توابع ساده تر در بسیاری از موارد دقت مورد نیاز را بر آورده میکند .

***به طور کلی در بخش سه به دلیل سخت تر بودن توابع (داشتن ابعاد بیشتر) دقت شبکه ی عصبی نسبت به بخش اول که توابع دو بعدی بودند کم تر است اما با تست کردن مقادیر و حالت های مختلف برای پارامتر ها (هر چه تا الان توضیح دادیم .) میتوان به دقت مطلوبی رسید .

بخش چهارم :

هدف :

در این بخش باید به کمک MLPRegressor یک تابع دلخواه یاد گرفته شود اما تفاوتش با بخش های قبلی در این است که در این جا تعدادی نقطه از یک تابع رسم شده (توسط خودمان) برای آموزش به شبکه ی عصبی میدهیم .

توضیح کد :

برای رسم کردن تابع و گرفتن نقاط آموزشی از آن ، ابتدا با رنگ سیاه خط خطی دلخواه رسم میکنیم .

سپس با رنگ آبی RGB =(0, 0, 255) قسمت بالای خط خطی را مشخص میکنیم .حالا از پکیج PIL ، Image را ایمپورت میکنیم و با Image.open('path').convert('RGB') عکس را باز میکنیم و به مدل RGB تبدیل میکنیم. (به این دلیل که میخواهیم قسمت آبی رنگ عکس را به عنوان تابع مشخص کنیم .) خروجی دستور بالا را به یک آرایه تبدیل میکنیم .که این آرایه به تعداد نقاط خط خطی خانه دارد که در هر خانه RGB ان نقطه مشخص شده است .(هر خانه سه بخش دارد که برای R و G و B است و به علاوه xو y آن نقطه هم در آن خانه مشخص شده اند.) حالا رنگ آبی را به شکل زیر تعریف میکنیم :

Blue = [0, 0, 255]

در نهایت با استفاده از کد زیر نقاطی که RGB ان ها با blue RGB یکسان است را پیدا میکنیم :

```
Y, X = np.where(np.all(im == blue, axis=2))
```

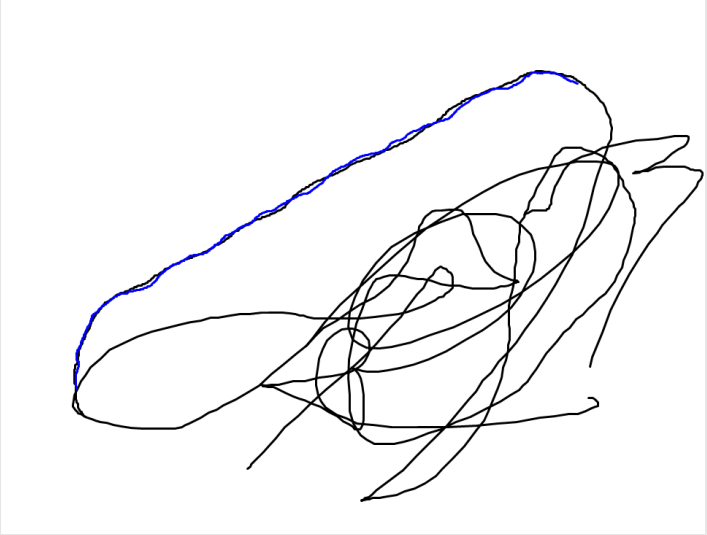
***باید به این نکته توجه کرد که x, y نقاط آرایه برعکس x, y عادی و مد نظر ما است .(این را با آزمون و خطا متوجه شدم) بنابراین در تکه کد بالا x های آرایه در y و y های آرایه در x ریخته شده است.

بقیه قسمت های کد مانند بخش اول است.

بررسی تاثیر پارامتر های مختلف :

1) میزان پیچیدگی تابع :

Function 1:



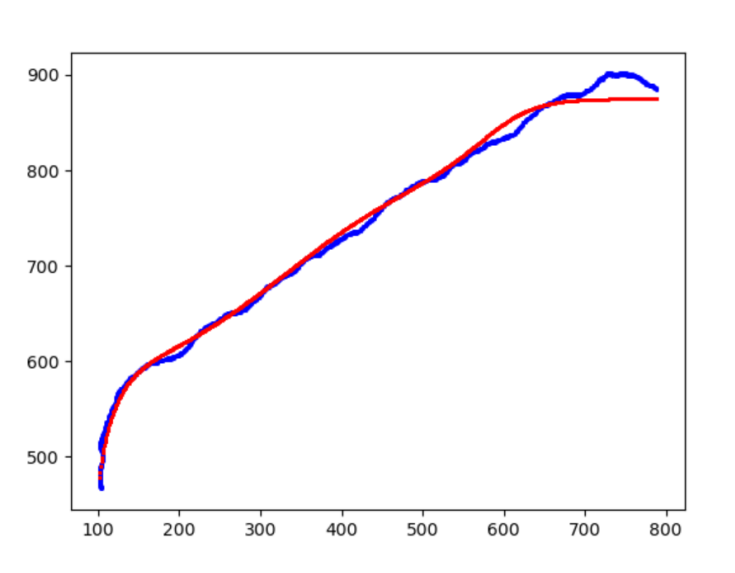
Iteration 4104, loss = 26.78733801

Iteration 4105, loss = 27.06647111

Iteration 4106, loss = 27.30396808

Iteration 4107, loss = 28.59980016

Iteration 4108, loss = 33.03380608



Function2 :

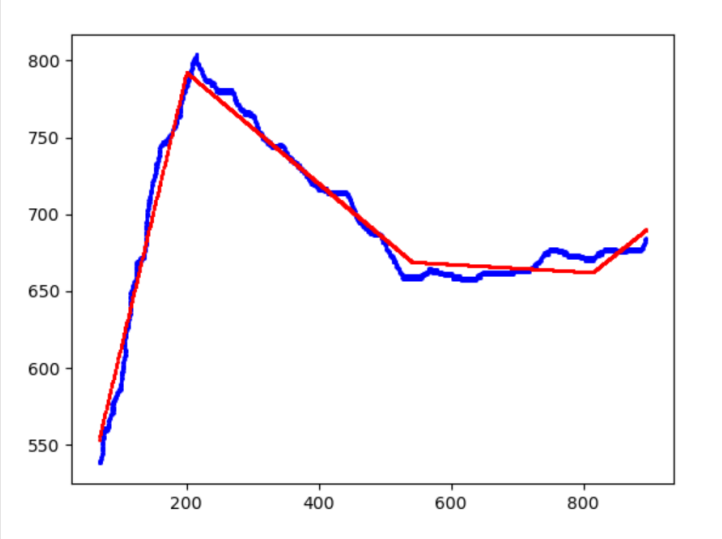
Iteration 886, loss = 46.90842226

Iteration 887, loss = 49.09237117

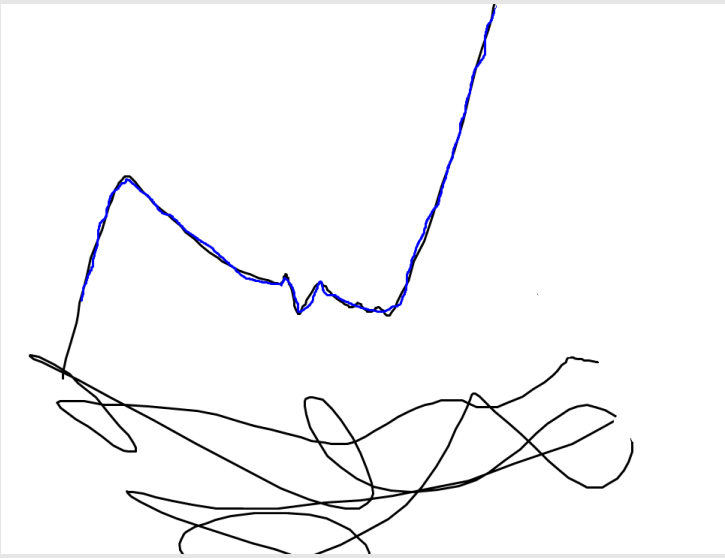
Iteration 888, loss = 50.12604327

Iteration 889, loss = 49.03692959

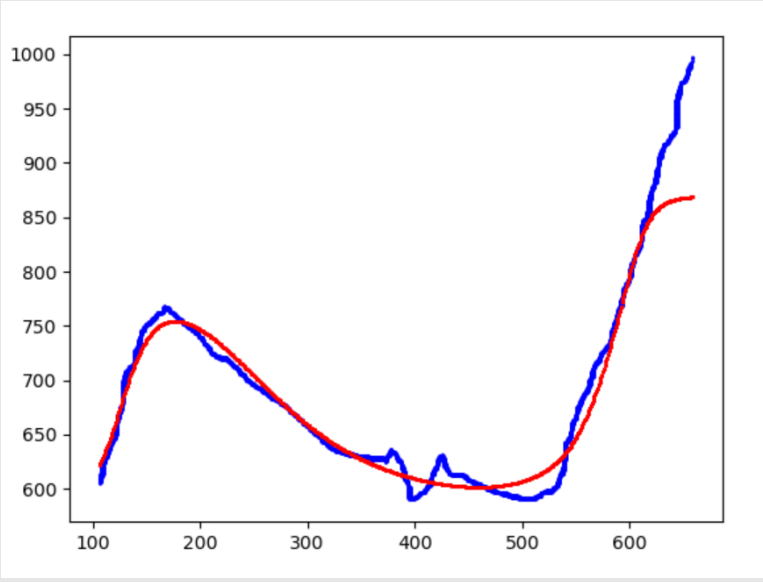
Iteration 890, loss = 47.74219953



Function3：



Iteration 3172, loss = 519.70952445
Iteration 3173, loss = 559.55095048
Iteration 3174, loss = 467.72995116
Iteration 3175, loss = 432.46043937
Iteration 3176, loss = 452.35393349



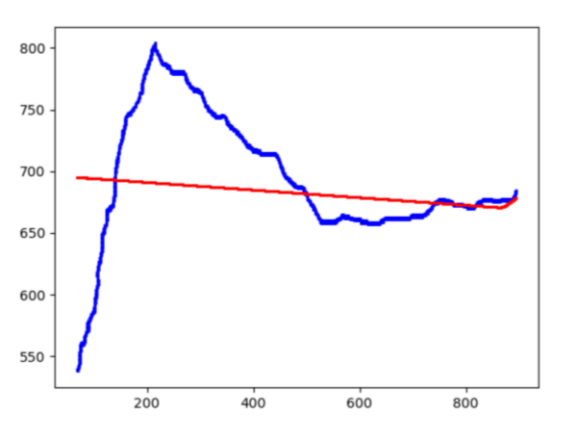
همان طور که مشخص است اگر تابع ساده باشد مانند تابع اول ، مقدار loss کم تر بوده و تابع پیشبینی شده به تابع واقعی نزدیک تر است . اما در تابع دوم مقدار loss تقریبا 1/5 برابر تابع اول است .در تابع سوم هم قله و دره های زیادی داریم که باعث میشود هم تعداد نورون های مورد نیاز بیشتر شود و هم این که مقدار loss مرحله آخر خیلی بیشتر از توابع 1 و 2 باشد.

2) تعداد لایه های شبکه و تعداد نورون های هر لایه :

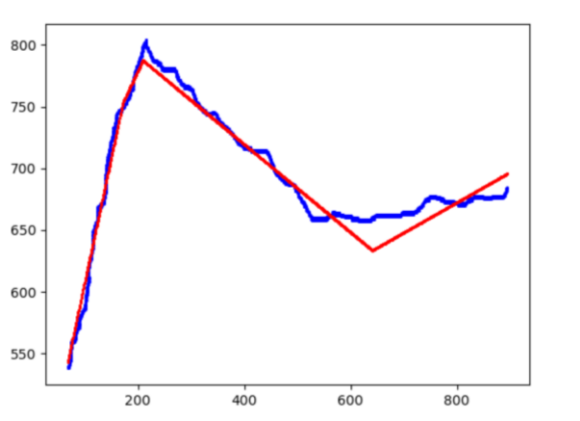
تاثیر این پارامتر ها را بر تابع دوم بررسی میکنیم :

Function2 :

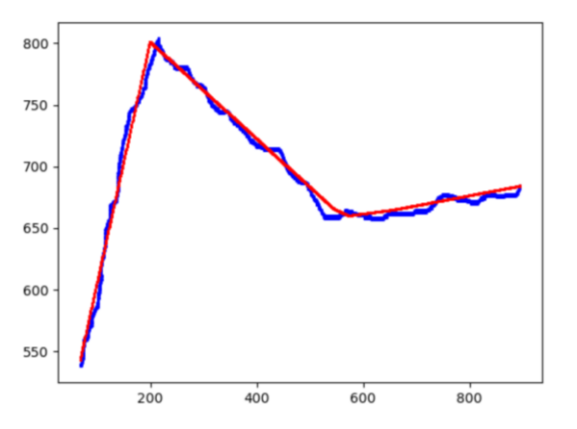
Hidden_layer_sizes = (20,) = 1 layer and 20 neurons



Hidden_layer_sizes =(10, 5) = 2 layers and 10 neurons in first layer and 5 neurons in second layer



Hidden_layer_sizes = (20, 10) = 2 layers and 20 neurons in first layer and 10 neurons in second layer



همان طور که مشخص است زیاد کردن تعداد لایه ها تاثیر بسیار زیادی بر دقت MLP گذاشته است و با این که مثلا در حالت اول 20 تا نورون و در حالت دوم 15 نورون داریم اما به دلیل این که در حالت دوم 2 لایه داریم MLP خیلی بهتر عمل کرده است.

3) تعداد چرخه ها :

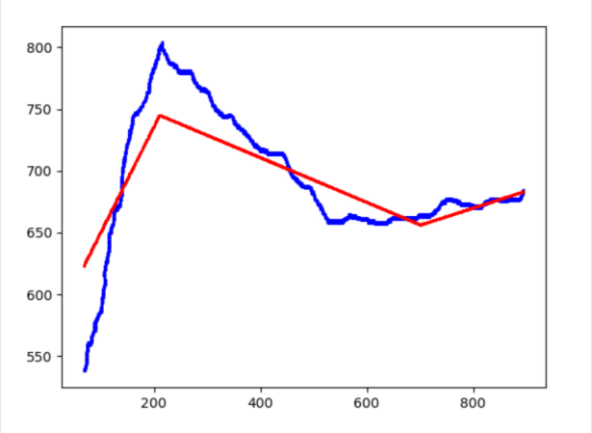
منطقا زیاد شدن تعداد چرخه ها تا یک جایی باعث بهتر شدن نتیجه میشود اما از یک جایی به بعد دیگر تاثیری ندارد .

برای بررسی این مورد تابع دوم را در نظر میگیریم :

Function2 :

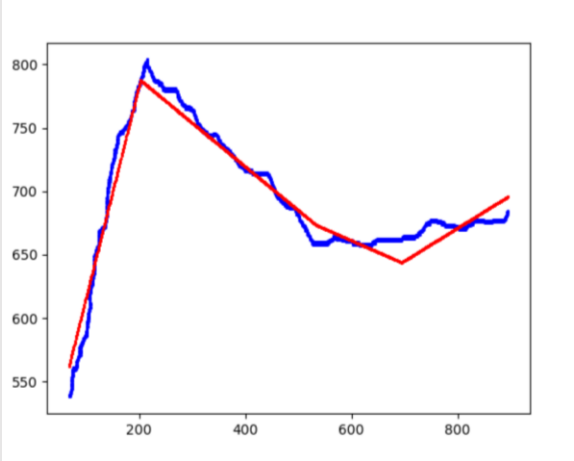
Max_iteration = 200

Iteration 295, loss = 460.31637960
Iteration 296, loss = 458.15258950
Iteration 297, loss = 453.98038603
Iteration 298, loss = 452.80532304
Iteration 299, loss = 451.13743866



Max_iteration = 800

Iteration 796, loss = 82.87845100
Iteration 797, loss = 85.21013924
Iteration 798, loss = 73.35508780
Iteration 799, loss = 72.93055164
Iteration 800, loss = 76.39929410



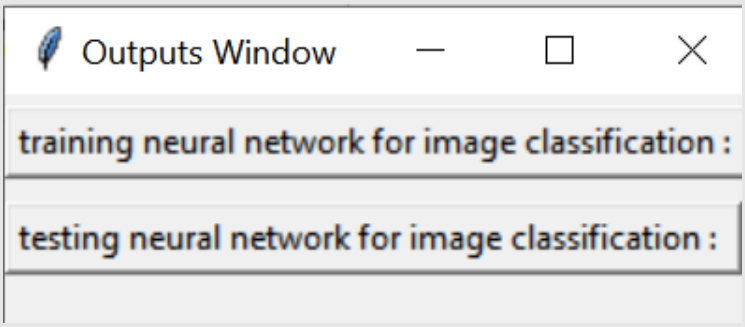
❊❊به طور کلی به نظر میرسد در صورتی که توابع مانند تابع 3 باشند زمان و تعداد نورون و لایه های بیشتری برای یادگیری تابع نیاز است . و بعضی از مواقع هم شبکه کاملا از بعضی از قله ها و دره ها را صرف نظر میکند .

بخش پنجم :

هدف :

هدف در این بخش کلاس بندی تعدادی تصویر از اعداد دست نویس انگلیسی است و بر خلاف بخش های قبلی این جا MLPClassifier استفاده میکنیم .

GUI این بخش :



توضیح کد :

در ابتدا عکس های اعداد را در متغیری به نام image_dataset ذخیره میکنیم . برای این کار از یک تابع استفاده میکنیم که تعدادی خروجی دارد که مهم ترین آن ها data و target است که data همان X ها هستند (عکس ها) و target ها نشان دهنده ی این است که هر عکس درواقع چه عددی را نشان میدهد . (برای مشخص کردن این موضوع هم تمام عکس های مربوط به هر عدد را در پوشه ای با نام همان عدد قرار دادم .)

سپس با استفاده از train_test_split دیتاست load شده را به دو دسته ی train و test تقسیم میکنیم . حالا یک MLPClassifier میسازیم که تمام ویژگی های MLPRegressor را دارد . سپس در max_iteration تعداد چرخه ها را مشخص میکنیم . حالا یک پنجره ایجاد میکنیم که با فشار دادن training neural network for image classification تابعی به نام fit_train_set اجرا میشود که در این تابع داده های آموزشی در max_iteration چرخه fit میشوند و شماره ی چرخه و مقدار خطای آن چرخه چاپ میشوند .

در نهایت بعد از تمام شدن آموزش شبکه با فشار دادن testing neural network for image classification ، تابع test اجرا میشود و در آن از تابع cross_validation برای محاسبه دقت مدل ایجاد شده بر روی داده های تست استفاده میشود.

بررسی تاثیر پارامتر های مختلف :

1) تعداد لایه های شبکه و تعداد نورون های هر لایه :

Hidden_layer_sizes = (10,)

…
iteration 97 :
error rate 0.311243
iteration 98 :
error rate 0.317185
iteration 99 :
error rate 0.322669
this will take a few minutes ...
Accuracy: 0.87 (+/- 0.03)

Hidden_layer_sizes = (10, 5)

…

iteration 97 :
error rate 0.274223
iteration 98 :
error rate 0.292505
iteration 99 :
error rate 0.279707
this will take a few minutes ...
Accuracy: 0.91 (+/- 0.04)

Hidden_layer_sizes = (20,)
…
iteration 97 :
error rate 0.185101
iteration 98 :
error rate 0.179616
iteration 99 :
error rate 0.176874
this will take a few minutes ...
Accuracy: 0.93 (+/- 0.02)

با توجه به 3 آزمایش بالا به نظر میرسد هر چه تعداد نورون ها بیشتر باشد دقت شبکه بیشتر میشود .

2) تعداد چرخه ها : به طور کلی زیاد شدن تعداد چرخه ها تا یک جایی باعث افزایش دقت میشود اما از یک جایی به بعد تاثیر زیادی در دقت شبکه ندارد .

Max_iteration = 10
…
iteration 7 :
error rate 0.213894
iteration 8 :
error rate 0.214808
iteration 9 :
error rate 0.210695
Accuracy: 0.92 (+/- 0.02)

Max_iteration = 30
...
iteration 27 :
error rate 0.201097
iteration 28 :
error rate 0.196984
iteration 29 :
error rate 0.201097
Accuracy: 0.93 (+/- 0.02)

Max_iteration = 100
…
iteration 97 :
error rate 0.185101
iteration 98 :
error rate 0.179616
iteration 99 :
error rate 0.176874

this will take a few minutes ...
Accuracy: 0.94 (+/- 0.01)

3) تابع فعالسازی :

Activation = ‘logistic’ :
…
iteration 47 :
error rate 0.231261
iteration 48 :
error rate 0.229890
iteration 49 :
error rate 0.225320
this will take a few minutes ...
Accuracy: 0.92 (+/- 0.02)

Activation = ‘relu’ :
…
iteration 47 :
error rate 0.197898
iteration 48 :
error rate 0.203839
iteration 49 :
error rate 0.206581
this will take a few minutes ...
Accuracy: 0.93 (+/- 0.02)

Activation = ‘tanh’ :
…
error rate 0.200183
iteration 47 :
error rate 0.196984
iteration 48 :
error rate 0.206581
iteration 49 :
error rate 0.204753

this will take a few minutes ...

Accuracy: 0.93 (+/- 0.02)

***با توجه به 3 آزمایش بالا به نظر میرسد که تابع فعالسازی logistic دقت کم تری برای شبکه ایجاد میکند و بهتر است که از relu یا tanh استفاده کنیم .

4) درصد داده های آموزشی و آزمایشی :

Test = 0.1 / train = 0.9 :

…

iteration 47 :

error rate 0.097260

iteration 48 :

error rate 0.106849

iteration 49 :

error rate 0.100000

this will take a few minutes ...

Accuracy: 0.93 (+/- 0.02)

Test = 0.3 / train = 0.7 :

…

iteration 47 :

error rate 0.201554

iteration 48 :

error rate 0.198812

iteration 49 :

error rate 0.199726

this will take a few minutes ...

Accuracy: 0.92 (+/- 0.02)

*** همانطور که از دو آزمایش بالا مشخص است (و البته انتظار هم میرود) ؛ هر چه تعداد داده های آموزشی نسبت به داده های آزمایشی بیشتر باشد دقت شبکه و میزان یادگیری آن بیشتر خواهد شد .

بخش ششم :

هدف :

تعدادی تصویر از اعداد دست نویس انگلیسی برداشته و مقداری noise به هر کدام از آن ها اضافه میکنیم و تعدادی از تصاویر noise دار به همراه تصاویر بدون noise آن ها به شبکه ی عصبی میدهیم تا یاد بگیرد دقیقا چه چیز هایی را باید از تصاویر حذف کند و چگونه تصاویر بدون noise را تولید کند . وقتی آموزش شبکه تمام شد باید تعدادی داده آموزشی به آن بدهیم و بررسی کنیم که تا چه حد میتواند noise را از این تصاویر حذف کند .

توضیح کد :

ابتدا data set را با استفاده از تابعی به نام load_image_files لود میکنیم و در image_dataset میریزیم .(این تابع همان تابع استفاده شده در بخش 5 است .) در این جا از آن جایی که به کلاس هر تصویر (مقدار عدد داخل تصویر) نیازی نداریم ، تنها data را از تابع میگیریم و این تصاویر را به دو دسته ی train و test تقسیم میکنیم .

در 3 خط زیر هم مقداری noise که با sigma مشخص میشود به داده ها اضافه میکنیم .

```
sigma = 0.3
noisy_train = np.clip(X_train + np.random.normal(0, sigma, X_train.shape), 0,
1)
noisy_test = np.clip(X_test + np.random.normal(0, sigma, X_test.shape), 0, 1)
```

سپس یک model برای یادگیری داده ها ایجاد میکنیم که دو تا لایه دارد و به شکل زیر تعریف شده است :

```
model = Sequential()
model.add(Dense(64, input_dim=64 * 64, activation='relu'))
model.add(Dense(64 * 64, activation='relu'))
model.compile(loss='mse', optimizer='adam')
```

سپس یک GUI ساده میسازیم که دو تا دکمه دارد اگر training neural network for image denoising را فشار دهیم تابع زیر اجرا میشود که در واقع داده های آموزشی را گرفته و آن ها را یاد بگیرد .(این که چگونه تصویر noise دار را به تصویر متناظر با آن که noise ندارد تبدیل کند .) :

```
model.fit(noisy_train, X_train, batch_size=512, epochs=100,
validation_split=0.2, verbose=2)
```

و اگر دکمه ی testing neural network for image denoising را فشار دهیم تابع زیر اجرا میشود که داده های آزمایشی را روی مدل ایجاد شده تست میکد و مقدار loss را خروجی میدهد.

```
model.evaluate(noisy_test, X_test)
```

بقیه ی کد صرفا انتخاب 30 تصویر رندم و نشان دادن تصویر اولیه ، تصویر noise دار و تصویری که شبکه عصبی noise آن را برطرف کرده است ، میباشد .

بررسی تاثیر مقدار noise ها مختلف :

Noise = 0.3

داده های آموزشی //...

Epoch 97/100

8/8 - 0s - loss: 0.0073 - val_loss: 0.0113

Epoch 98/100

8/8 - 0s - loss: 0.0072 - val_loss: 0.0113

Epoch 99/100

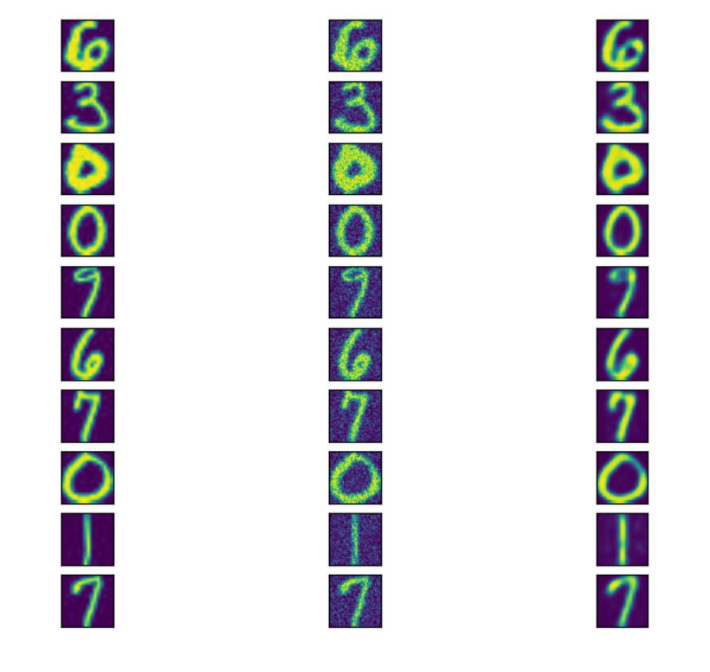
8/8 - 0s - loss: 0.0072 - val_loss: 0.0112

Epoch 100/100

8/8 - 0s - loss: 0.0071 - val_loss: 0.0112

دقت در داده های آموزشی //

69/69 [==============================] - 0s 3ms/step - loss: 0.0108



*** ستون اول از چپ نشان دهنده ی داده های اولیه ، ستوان دوم نشان دهنده ی داده هایی که به آن ها noise اضافه شده اند و ستون آخر از چپ نشان دهنده ی داده های رفع noise شده توسط شبکه ی عصبی است .

Noise = 0.5

داده های آموزشی //...

Epoch 97/100

8/8 - 0s - loss: 0.0066 - val_loss: 0.0136

Epoch 98/100

8/8 - 0s - loss: 0.0065 - val_loss: 0.0137

Epoch 99/100

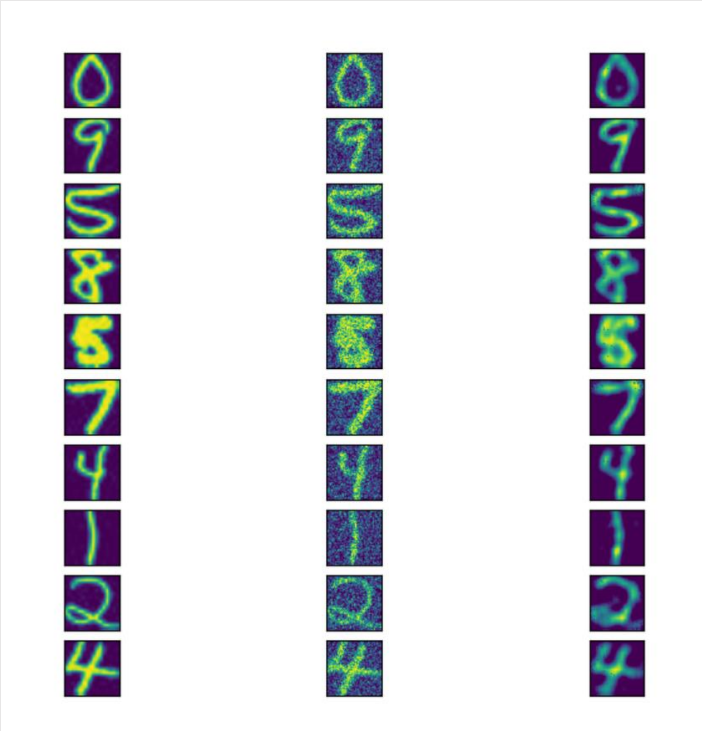
8/8 - 0s - loss: 0.0064 - val_loss: 0.0137

Epoch 100/100

8/8 - 0s - loss: 0.0064 - val_loss: 0.0137

دقت در داده های آزمایشی //

69/69 [==============================] - 0s 3ms/step - loss: 0.0136



Noise = 0.7

داده های آموزشی //...

Epoch 97/100

8/8 - 0s - loss: 0.0077 - val_loss: 0.0196

Epoch 98/100

8/8 - 0s - loss: 0.0076 - val_loss: 0.0196

Epoch 99/100

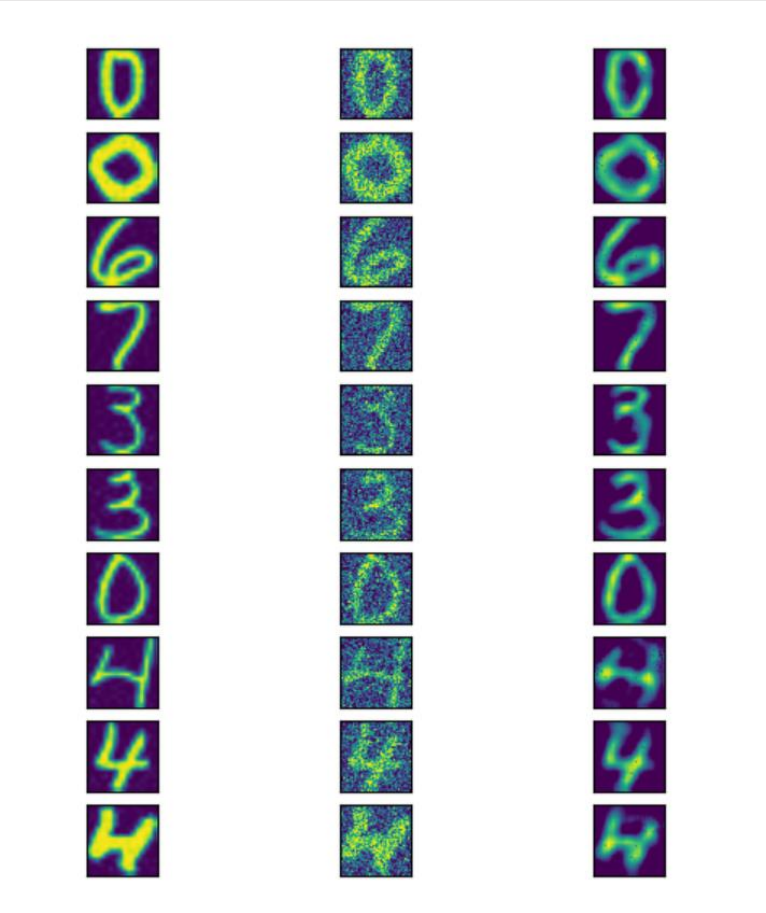
8/8 - 0s - loss: 0.0076 - val_loss: 0.0196

Epoch 100/100

8/8 - 0s - loss: 0.0076 - val_loss: 0.0197

دقت در داده های آزمایشی//

69/69 [==============================] - 0s 3ms/step - loss: 0.0197



*** با توجه به سه آزمایش بالا متوجه میشویم هر چقدر مقدار noise بیشتر میشود مقدار loss بیشتر میشود یعنی دقت یادگیری کاهش پیدا میکند .