

بسم الله الرحمن الرحيم

گزارش تمرین سری ۵ پایتون نوروساینس

امیرحسین رستمی 96101635

پوریا ملاحسینی 96102459

دکتر کربلایی آقاجان

دانشگاه صنعتی شریف

98 بهار

سوال اول: مقدمات AND و XOR

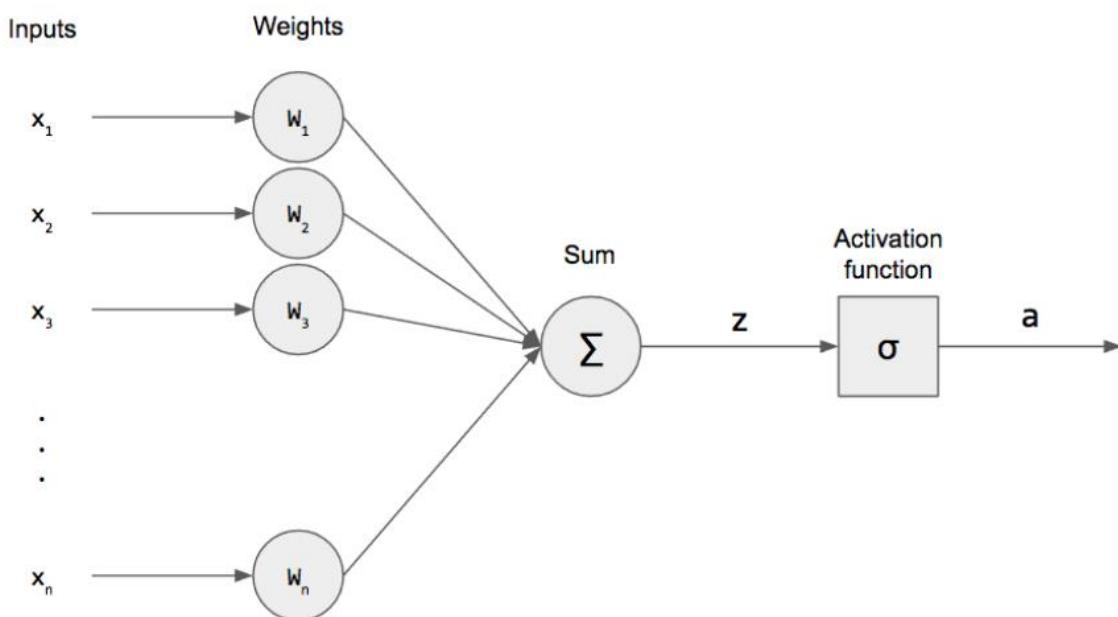
بخش اول: ابتدا که به کمک راهنمایی های داده شده فرم ورک پایتورچ را به Anaconda وارد می کنیم و از صحت عملکرد آن از به کمک دستورات کامند لاین مطمئن می شویم.

```
/**/
```

بخش دوم و بخشی از بخش ششم (علت چند لایه بودن XOR)

در ابتدا به توضیح مفصل نحوه ای حل و یادگیری می پردازیم و هم چنین

Learning “Learning”!!!



I found it difficult understanding how the Perceptron works with Logic gates (AND, OR, NOT, and so on). I decided to check online resources, but as of the time of writing this, there was really no explanation on how to go about it. So after personal readings, I finally understood how to go about it and now I am telling U what to do ...

First, we need to know that the Perceptron algorithm states that:

Prediction (y') = 1 if $Wx+b \geq 0$ and 0 if $Wx+b < 0$

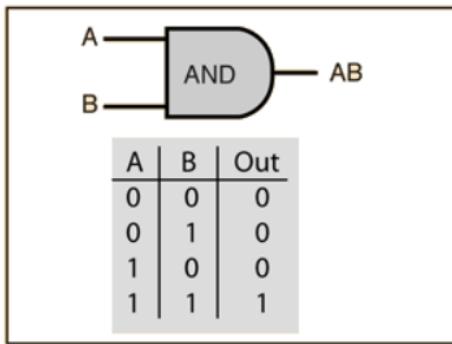
Also, the steps in this method are very similar to how Neural Networks learn, which is as follows;

- Initialize weight values and bias
- Forward Propagate
- Check the error
- Back propagate and Adjust weights and bias
- Repeat for all training examples

Now that we know the steps, let's get up and running: (AND, XOR Gates)

AND Gate

From our knowledge of logic gates, we know that an AND logic table is given by the diagram below



AND Gate

The question is, what are the weights and bias for the AND perceptron?

First, we need to understand that the output of an AND gate is 1 only if both inputs (in this case, x_1 and x_2) are 1. So, following the steps listed above;

Row 1

- From $w_1x_1+w_2x_2+b$, initializing w_1 , w_2 , as 1 and b as -1, we get;

$$x_1(1)+x_2(1)-1$$

- Passing the first row of the AND logic table ($x_1=0$, $x_2=0$), we get;

$$0+0-1 = -1$$

- From the Perceptron rule, if $Wx+b < 0$, then $y' = 0$. Therefore, this row is correct, and no need for Backpropagation.

Row 2

- Passing ($x_1=0$ and $x_2=1$), we get;

$$0+1-1 = 0$$

- From the Perceptron rule, if $Wx+b \geq 0$, then $y' = 1$. This row is incorrect, as the output is 0 for the AND gate.
- So we want values that will make the combination of $x_1=0$ and $x_2=1$ to give y' a value of 0. If we change b to -1.5, we have;

$$0+1-1.5 = -0.5$$

- From the Perceptron rule, this works (for both row 1, row 2 and 3).

Row 4

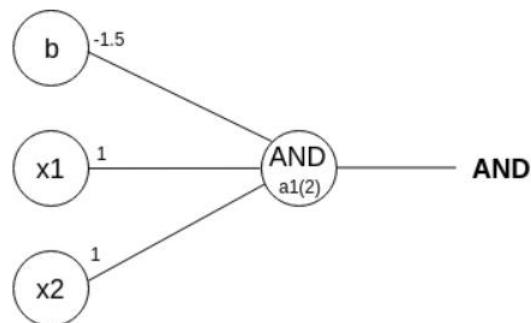
- Passing ($x_1=1$ and $x_2=1$), we get;

$$1+1-1.5 = 0.5$$

- Again, from the perceptron rule, this is still valid.

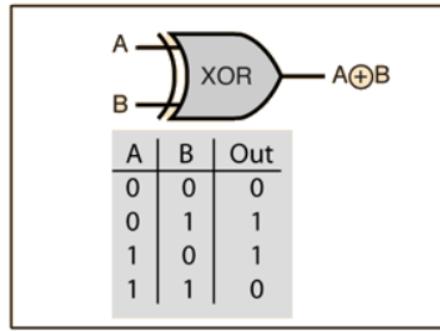
Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is;

$$x_1+x_2-1.5$$



Now let go to implement XOR gate ...

XOR Gate:



XOR Gate

The Boolean representation of an XOR gate is;

$$x_1x_2 + x_1x_2$$

We first simplify the Boolean expression:

$$x_1x_2 + x_1x_2 + x_1x_1 + x_2x_2$$

$$x_1(x_1 + x_2) + x_2(x_1 + x_2)$$

$$(x_1 + x_2)(x_1 + x_2)$$

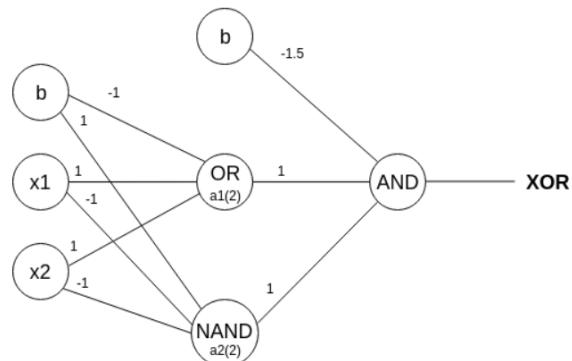
$$(x_1 + x_2)(x_1x_2) \rightarrow ' stands for prim...$$

From the simplified expression, we can say that the XOR gate consists of an OR gate ($x_1 + x_2$), a NAND gate ($-x_1-x_2+1$) and an AND gate ($x_1+x_2-1.5$).

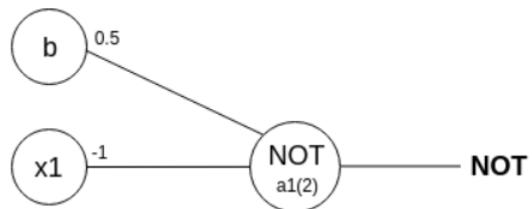
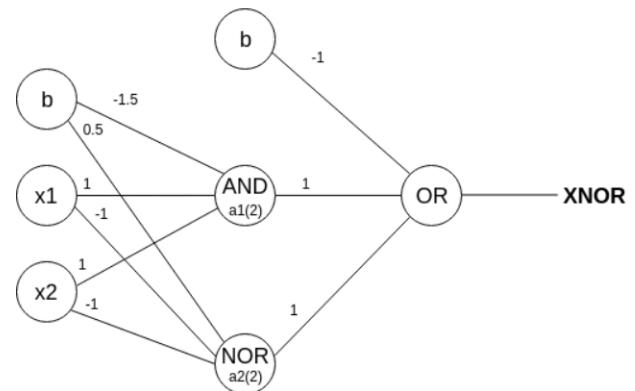
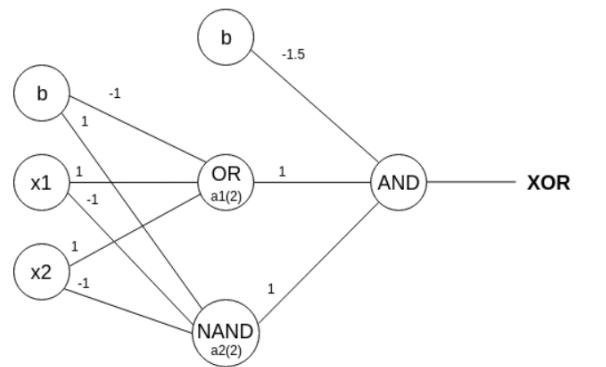
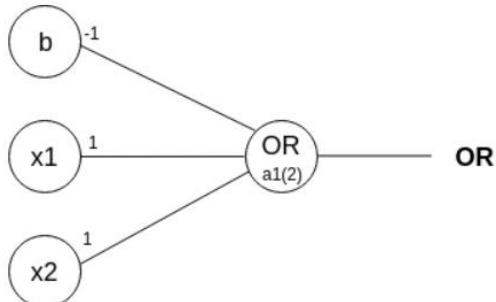
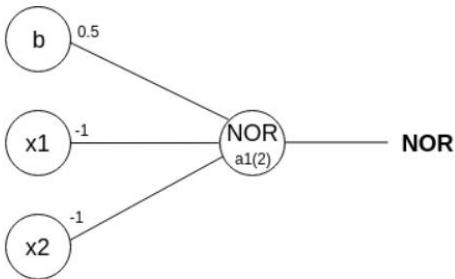
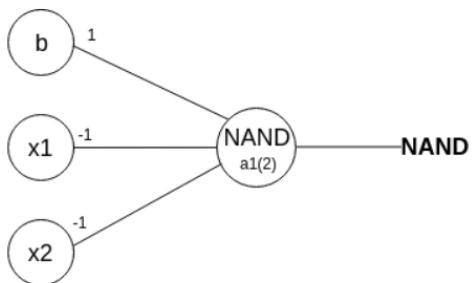
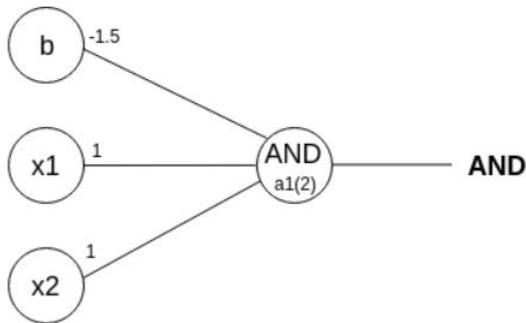
This means we will have to combine 2 perceptions:

NOTE: As you see below I used OR and NAND gates which their implementations is similar to the method I told above for implementing the AND Gate ... So I refused to tell the completely...

- OR (x_1+x_2-1)
- NAND ($-x_1-x_2+1$)
- AND ($x_1+x_2-1.5$)



Conclusion: let's see all Gates graphs All together ...



بخش سوم:

البته پیش از توضیح کد لازم است اشاره کنم که در کل این پروژه بنده از گیت هاب های زیر به کرات استفاده کردم و برای اینکه خطای خطا ها نیز اپتیمایز بشوند (یعنی حول یک مرکز خطای خطا های با واریانس زیاد دور ریخته شود و داده های تمیز تر گرد هم جمع شوند از توابع خیلی مفید زیر که در گیت های استفاده و توضیح داده شده است استفاده نمودم) توابع از قبیل: optimizer و ...

- <https://github.com/jcjohnson/pytorch-examples>
- <https://github.com/synxlin/nn-compression>

در این سوال یک شبکه عصبی با یک لایه عصبی به صورت یک کلاس مجزا تعریف کردم و کانستراکتور این کلاس فیلد های زیر را می گیرد: (به کمک بررسی گیت های متعدد به اینکه همه ی فیلد های مهم داخل کانستراکتور باشد و از خارج اصلاح نگردد رسیدم!)

```
class NeuralNetwork(nn.Module):
```

```
    def __init__(self, inSize, outSize, hidSize, learningRate):
```

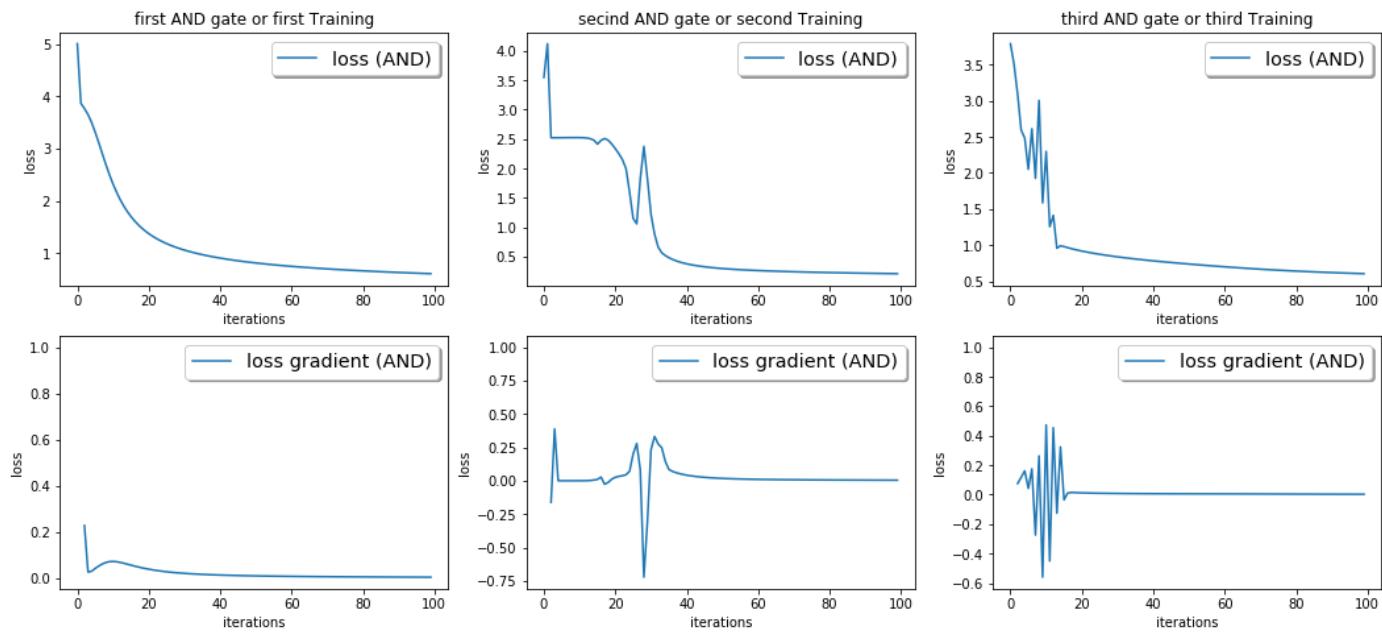
هم چنین از آنجا که در هر مرحله نیاز است که داده ها و نمودار ها از روی آنها کشیده شود داخل قسمتی به نام Log loss که در هر مرحله Data و هم چنین Neuron Data ذخیره می گردد و به کمک توابع getLossData این مقادیر گرفته می شوند.

طبق گفته های داک ارایه شده برای یادگیری بهتر بعد ورودی را افزایش می دهیم و یک مولفه ی 1 اضافی در نظر میگیریم. و هم چنین به کمک جدول صحبت epoch ها رو تعلیم می دهیم که این عمل تعلیم به کمکتابع train انجام می شود.

پیاده سازی توابع forward و backward هم به کرات در داک های مختلف و هم چنین کلاس تی ای گفته شده است لذا از ذکر مجدد این مطالب خودداری کرده و رجوع به کد را کافی می دانم.

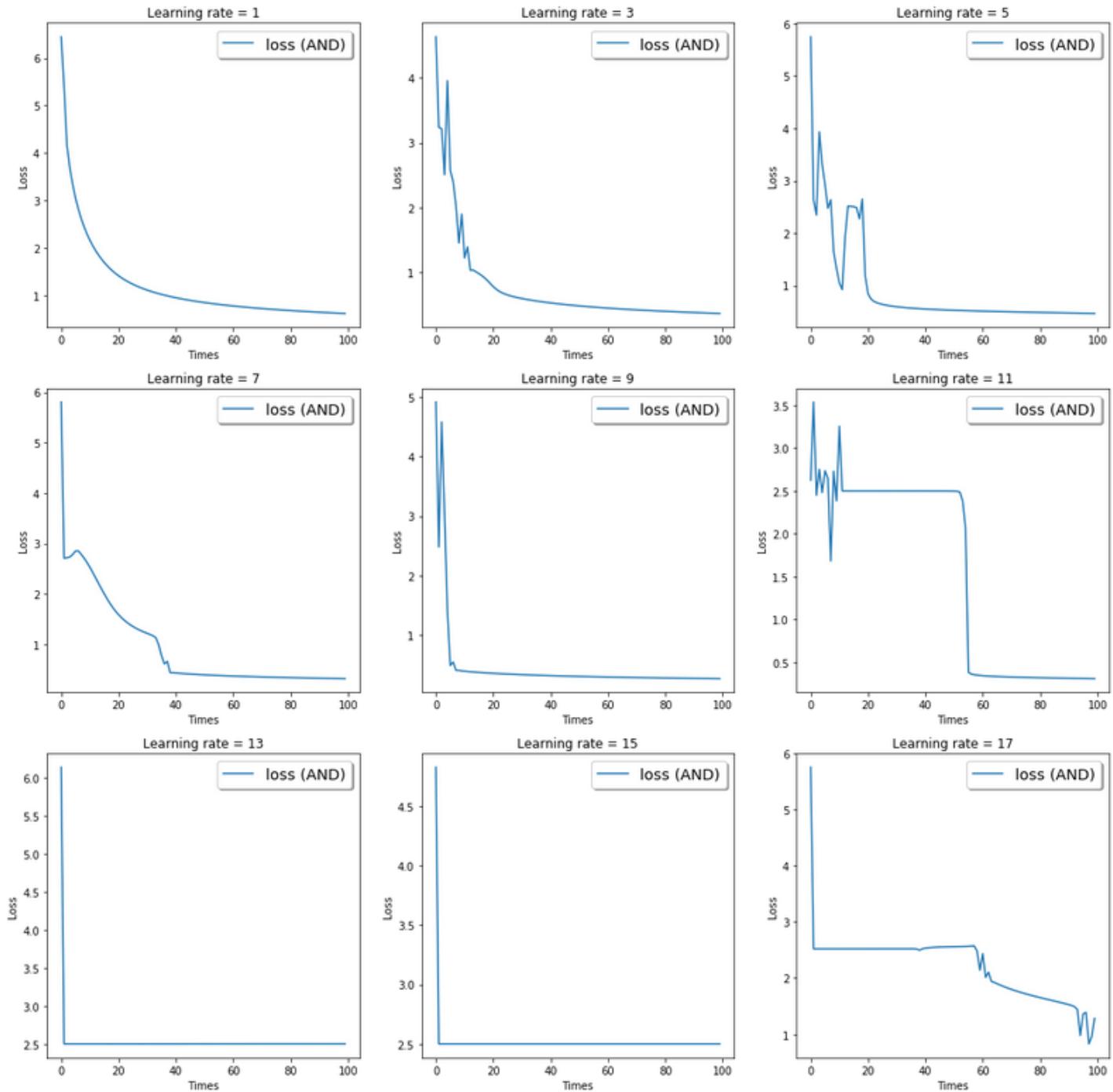
و پیش فرض عملکردی ما ثابت بودن نرخ یادگیری می باشد. البته این را می توان تغییر داد و البته در قسمت اول بخش چهارم همین کار رو می کنیم.

نمودار های بخش سوم: برای این قسمت بنده سه نمونه گیت And train کردن یک گیت And را انجام داده
ام و حاصل نمودار های زیر گردیده است:



بخش چهارم:

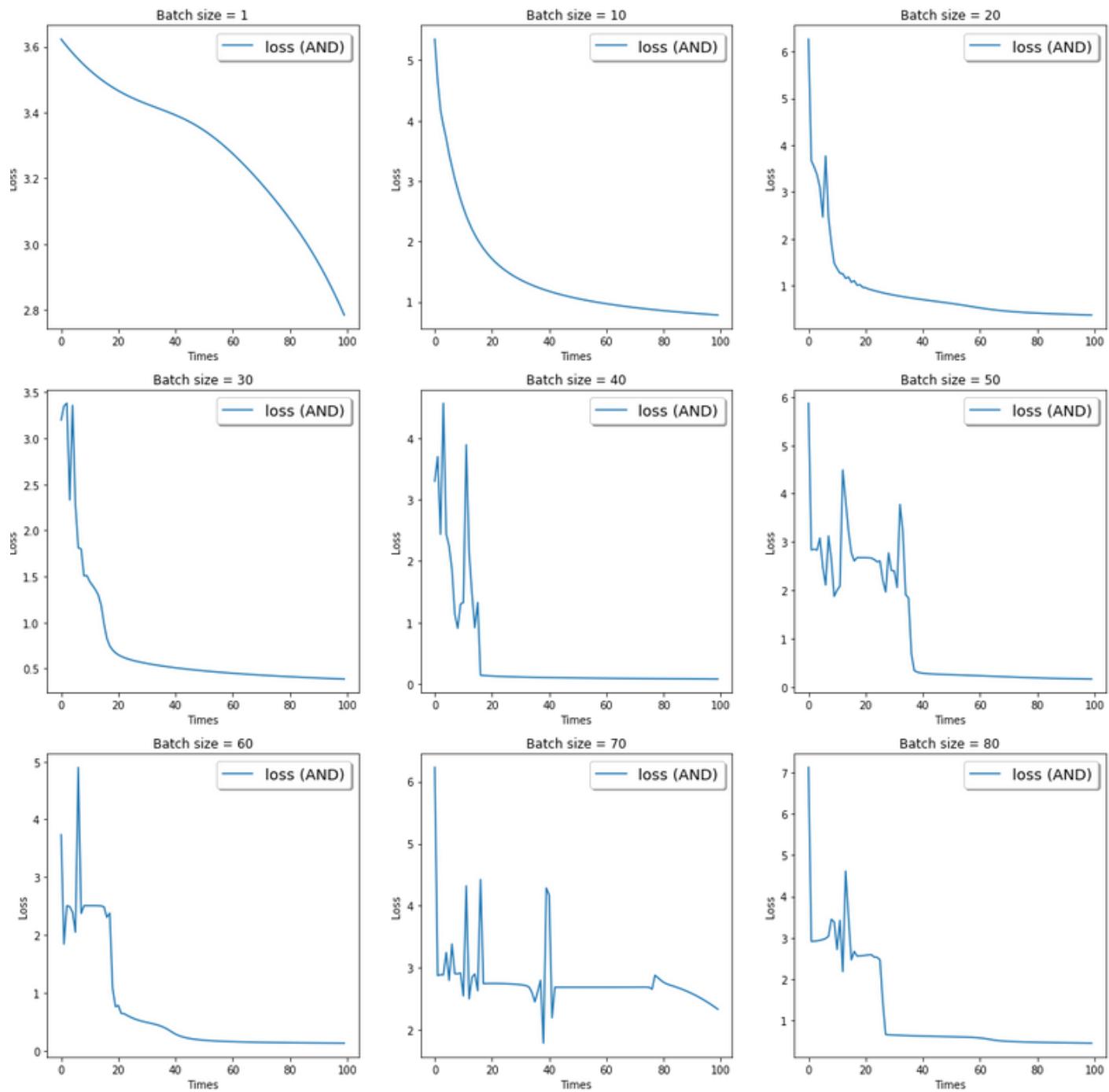
- قسمت اول: در این بخش اثر **تغییرات learning rate** را بررسی می کنیم و داریم که:
 - در این قسمت learning Rate ها به ترتیب اعداد فرد تا 17 می باشند و البته ذکر یک نکته حائز اهمیت است و آن این است که **افزایش خیلی بیش از حد learning Rate** خیلی خوب نمی باشد و گاهای باعث واگرایی نیز می گردد.



نتیجه گیری: مشخص است که نرخ یادگیری هر چقدر زیاد باشد داریم که همگرایی به مقدار نهایی زودتر رخ می دهد و این خود به دلیل بیشتر شدن تعداد گام های برداشته شده در هر مرحله و اندازه ی گام ها می باشد. و توجه کنید که در هر چقدر مقدار نرخ یادگیری کوچک باشد داریم که خطای زیاد تر می شود چرا که نمی تواند خطای زیر حد نرخ یادگیری را تغییر دهد.

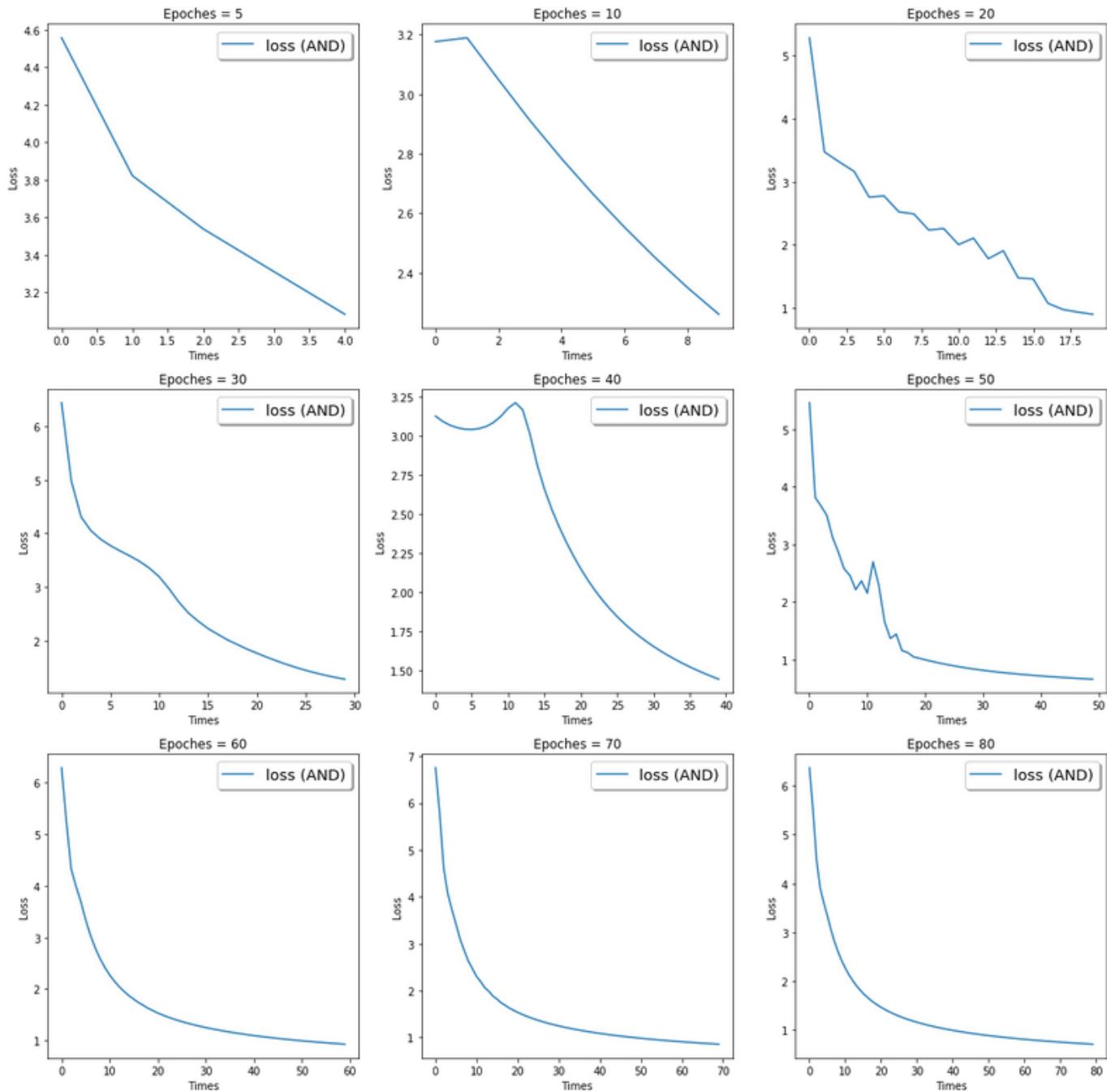
- البته گاهی وقتا اگر بیش از حد نرخ یادگیری را تغییر بدھیم واگرا می شود!!! که در ادامه مثال هایی در این مورد در سوال های بعدی زده شده است.

قسمت دوم: در این بخش اثرات تغییر batch size را بررسی می کنیم و داریم که:



نتیجه گیری: در فرآیند یادگیری هر بار بسته‌ی داده‌ها به شبکه داده می‌شود و هر چقدر این بسته‌ی داده‌ها بزرگ باشند داریم که سیر خطا زودتر همگرا به صفر می‌گردد. (چراکه وقتی سایز بسته‌ها بیشتر باشد میزان داده‌های در کنار هم برای ترین شبکه افزایش یافته و شبکه بهتر یاد می‌گیرد و خطا حاصل کاهش می‌یابد...)

قسمت سوم: بررسی اثر تغییرات number of epochs و داریم که نمودار به شرح زیر است:



نتیجه گیری:

در فرآیند یادگیری داریم که با افزایش تعداد epoch ها میزان و سیر همگرا شدن loss به صفر بیشتر می گردد و این پارامتر یک ویژگی متمایز با سایر پارامتر های شبکه ای عصبی دارد و آن این است که این پارامتر هر چقدر زیاد تر شود داریم که کمک بیشتری

به همگراشدن شبکه می کند و بدون اینکه زیادتر شدن بیش از حد آن برخلاف مثلا (Rate or number of hidden layers ... باعث واگرایی شبکه شود و این خود امتیاز مشتبی است که این ویژگی شبکه دارد.

بخش پنجم:

ابتدا ذکر یک نکته مهم است:

در این قسمت بندۀ در نظر گرفتم که بیایم یک بار با `relu` و یک بار با `sigmoid` شبکه را آموزش بدهم و لذا نیاز بود که تغییر کوچکی در شبکه قسمت قبل بدهم و این تغییر را به کمک **وراثت** انجام دادم.

لازم به ذکر است که در همه قسمت های قبل از تابع `sigmoid` استفاده شده است و از آنجا که فعالیت نورون ها تحت تاثیر عوامل متعدد است لذا خواستم اثر تغییر تابع اکتیویشن را در این مورد نیز ببینیم لذا کد را اندکی تغییر دادم.

(Activation Function)

```
netNeuron1_1 = NetNeuron(3, 1, 10, 10, 'sigmoid') # AND with 10 hidden layers sigmoid function  
netNeuron1_2 = NetNeuron(3, 1, 1, 10, 'sigmoid') # AND with one hidden layer sigmoid function  
netNeuron2_1 = NetNeuron(3, 1, 10, 10, 'relu') # AND with 10 hidden layer and relu function  
netNeuron2_2 = NetNeuron(3, 1, 1, 10, 'relu') # AND with one hidden layer and relu function
```

نمودار اول : 10 لایه پنهان با تابع سیگموئد

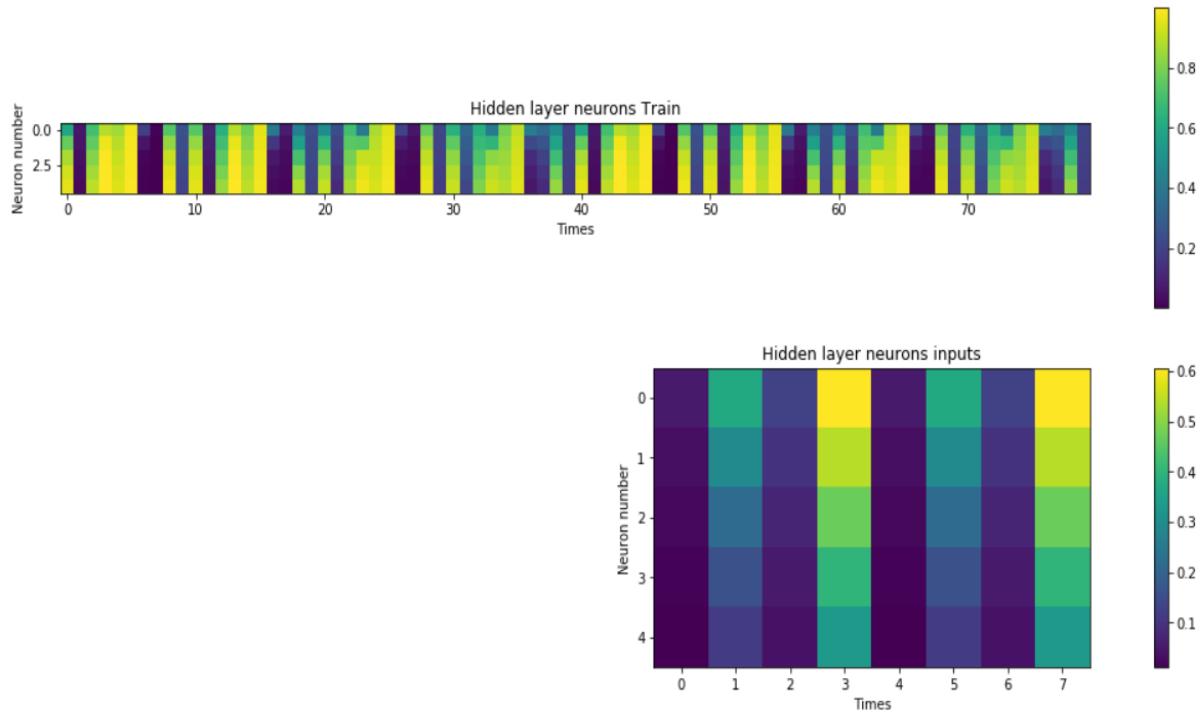
نمودار دوم : 1 لایه پنهان با تابع سیگموئد

نمودار سوم : 10 لایه پنهان با تابع رل

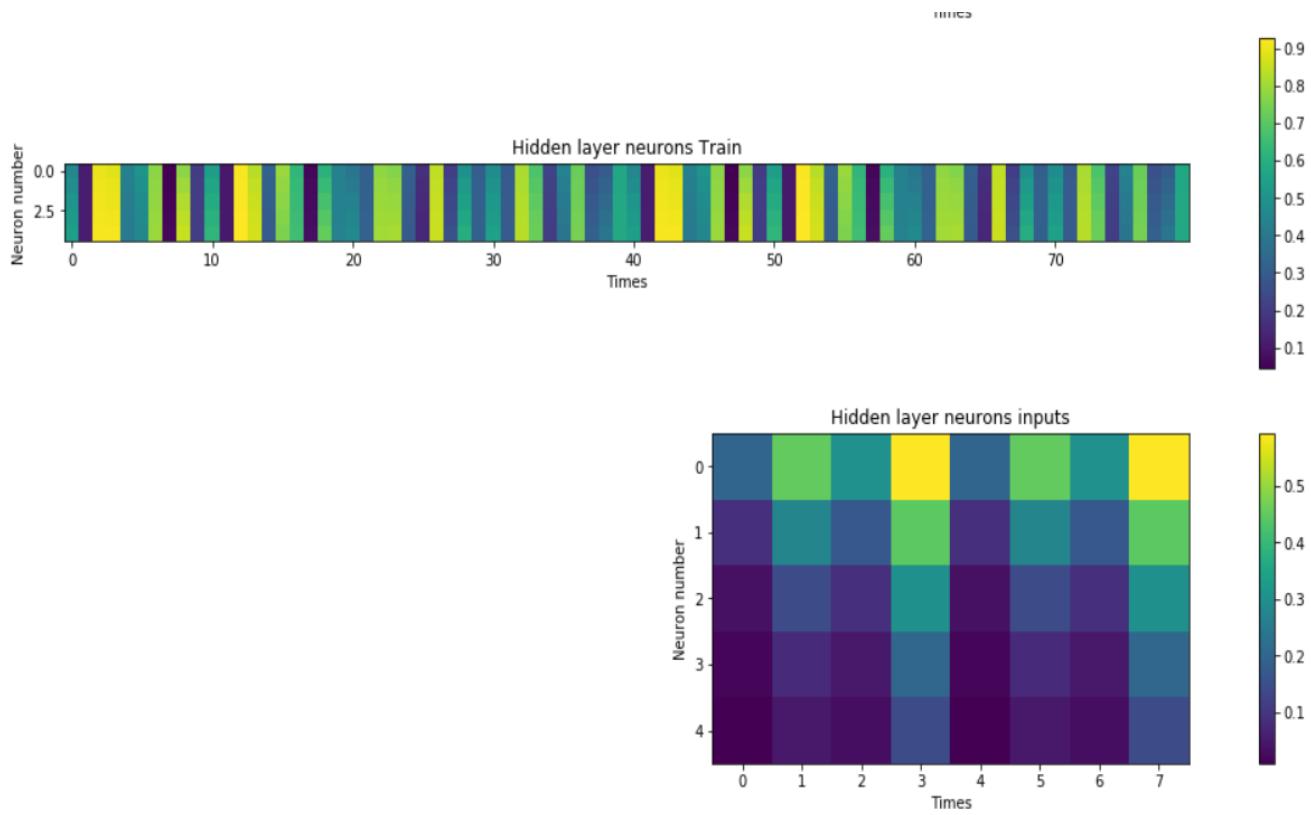
نمودار چهارم : یک لایه پنهان با تابع رل

- نتیجه گیری : توجه کنید که نمودار های صفحه ی بعد نشان می دهید که چقدر فعالیت نورون ها به مقادیر وزن ها در ابتدا بستگی دارد و داریم که در اصل می دانیم که تعداد نورون ها زیاد می باشد و داریم که عملکرد و فعالیت شبکه در نهایت منتها به همین فعالیت های نورون ها دارد و لذا داریم که تغییرات در وزن ها باعث تغییرات در فعالیت نورون ها می گردد و افزایش یافتن یک وزن منتها به فعال تر شدن فعالیت نورون مرتبط با آن وزن می گردد حال آنکه کمتر شدن یک وزن حین آموزش منجر به کاهش فعالیت نورون مربوط به آن وزن می گردد و داریم که این تغییر فعالیت ها به کمک رنگ ها و transient رنگ ها در نمودار های صفحه ی بعد مشخص تر است.

Sigmoid Activation Function ...



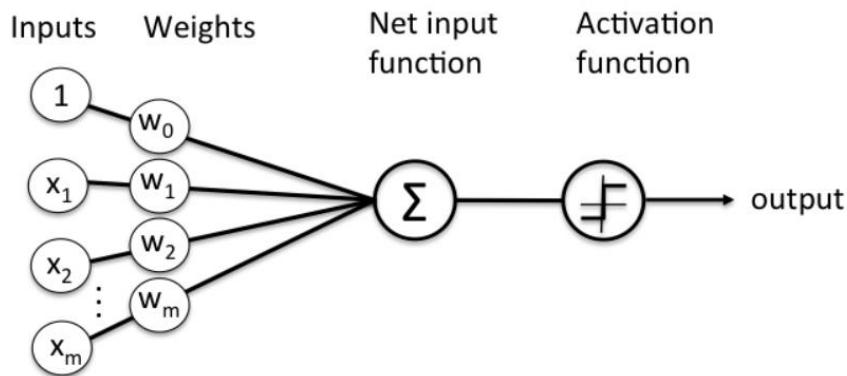
Relu Activation Function ...



بخش ششم:

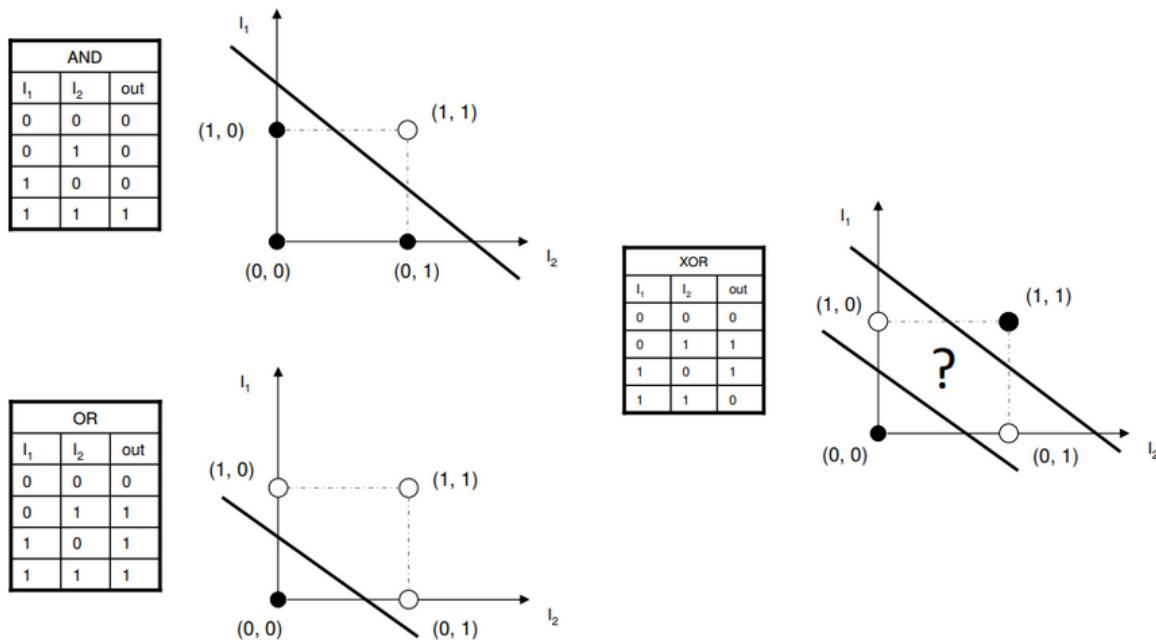
- علت اینکه XOR چند لایه است را در قسمت اول در کنار and ذکر کرده ام لذا از ذکر مجدد آن پرهیز می کنیم صرفا خلاصه ای از آن را ذکرمی کنیم:

The perceptron is a model of a hypothetical nervous system originally represented by the schematic shown in the figure below.



As we can see, it calculates a weighted sum of its inputs and thresholds it with a step function. Geometrically, this means the perceptron can separate its input space with a hyperplane. That's where the notion that a perceptron can only separate linearly separable problems came from. Since the XOR function is not linearly separable, it really is impossible for a single hyperplane to separate it.

Look:

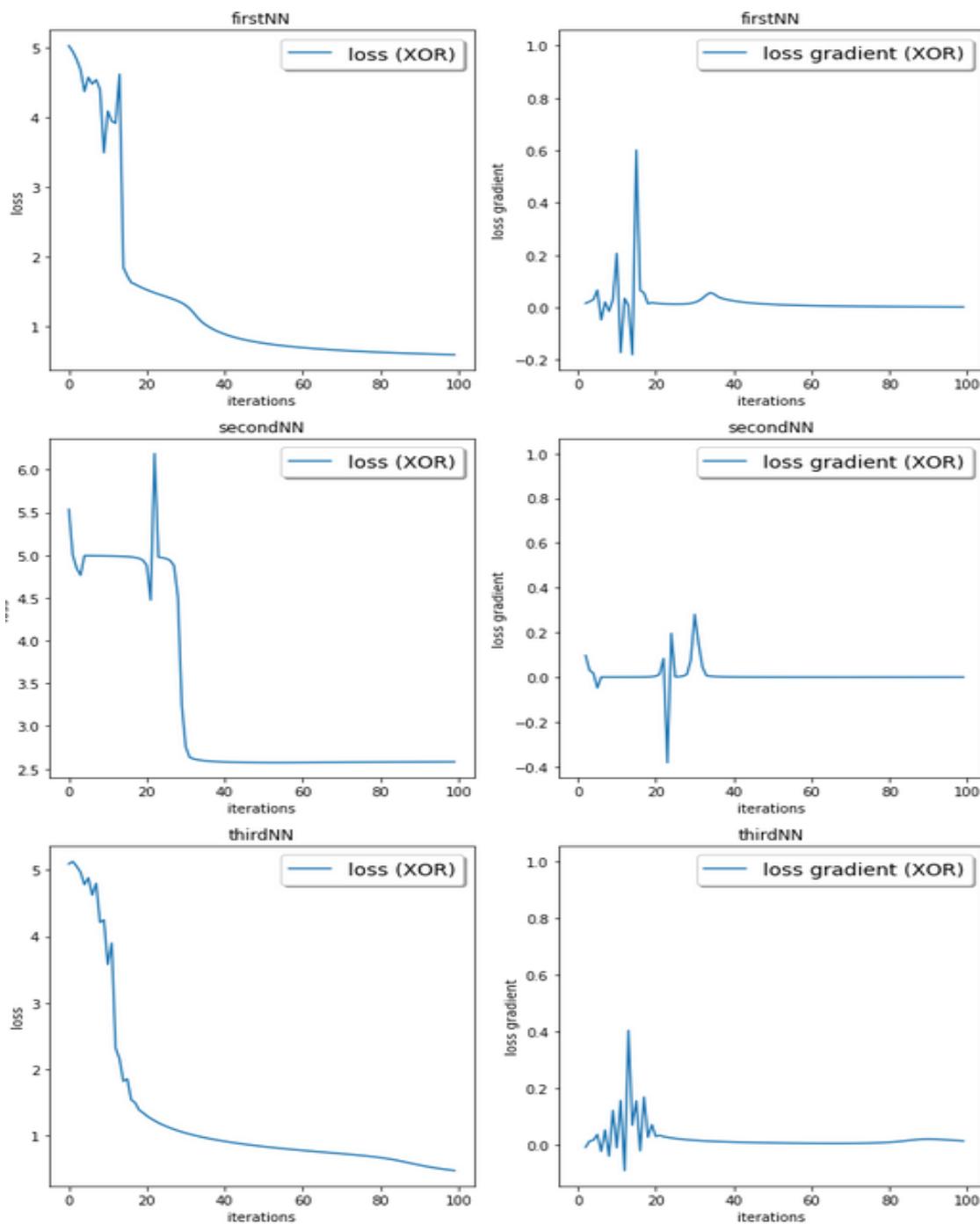


همانطورکه از شکل فوق پیداست مساله XOR به کمک تک لایه قابل learn نیست.

حال داریم که همانند مطالب گفته شده در بخش اول مساله XOR را حل کرده و کد مربوطه را با تعداد لایه های بیشتر از یک می زنیم. (همه کد ها در فایل zip ضمیمه شده است).

در این بخش سه نمونه مثال از XOR آورده ام:

firstNN, secondNN and thirdNN with 4,8,12 hiddenSizes ...



بخش هفتم قسمت اول:

من در این قسمت شبکه ای جدید طراحی کردم که یک فیلد کانسٹراکتور آن تعداد لایه های میانی را تعیین می کند و در نظر گرفتم که تعداد لایه ها بین 1 تا 6 می باشد.

و روش بندۀ در زیاد تر کردن لایه به شکل زیر است که : (این کد از قسمت forward استخراج شده است)

Increasing number of layers via nested Ifs ...

```
def __init__(self, input_size, output_size, hidden_size, numberOfLayers):

    // from forward part ...

    if self.layersNum == 1:

        out = self.inputWeights(x)
        out = self.activation_function(out)

    elif self.layersNum == 2:

        out = self.inputWeights(x)
        out = self.activation_function(out)

        out = self.fromFirstLayerTOutWeights(out)
        out = self.activation_function(out)

    elif self.layersNum == 3:

        out = self.inputWeights(x)
        out = self.activation_function(out)

        out = self.fromFirstLayerTSecondLayerWeights(out)
        out = self.activation_function(out)

        out = self.fromSecondLayerTOutWeights(out)
        out = self.activation_function(out)
```

البته تى اى محترم روش دیگری در افزایش تعداد لایه ها فرمودند ولیکن که از آنجا که من به کمک روش فوق کد هارا پیاده سازی کردم این روش را بیان کردم.(البته این قطعه کد بخشی از مراحل افزایش لایه ها می باشد).

Creating matrixes which create correlation between inputs and outputs

```
def setNetConfigMat(self):

    if self.layersNum == 1:
        self.inputWeights = nn.Linear(self.inputSize, self.outputSize) # i X h tensor in to hidden 1

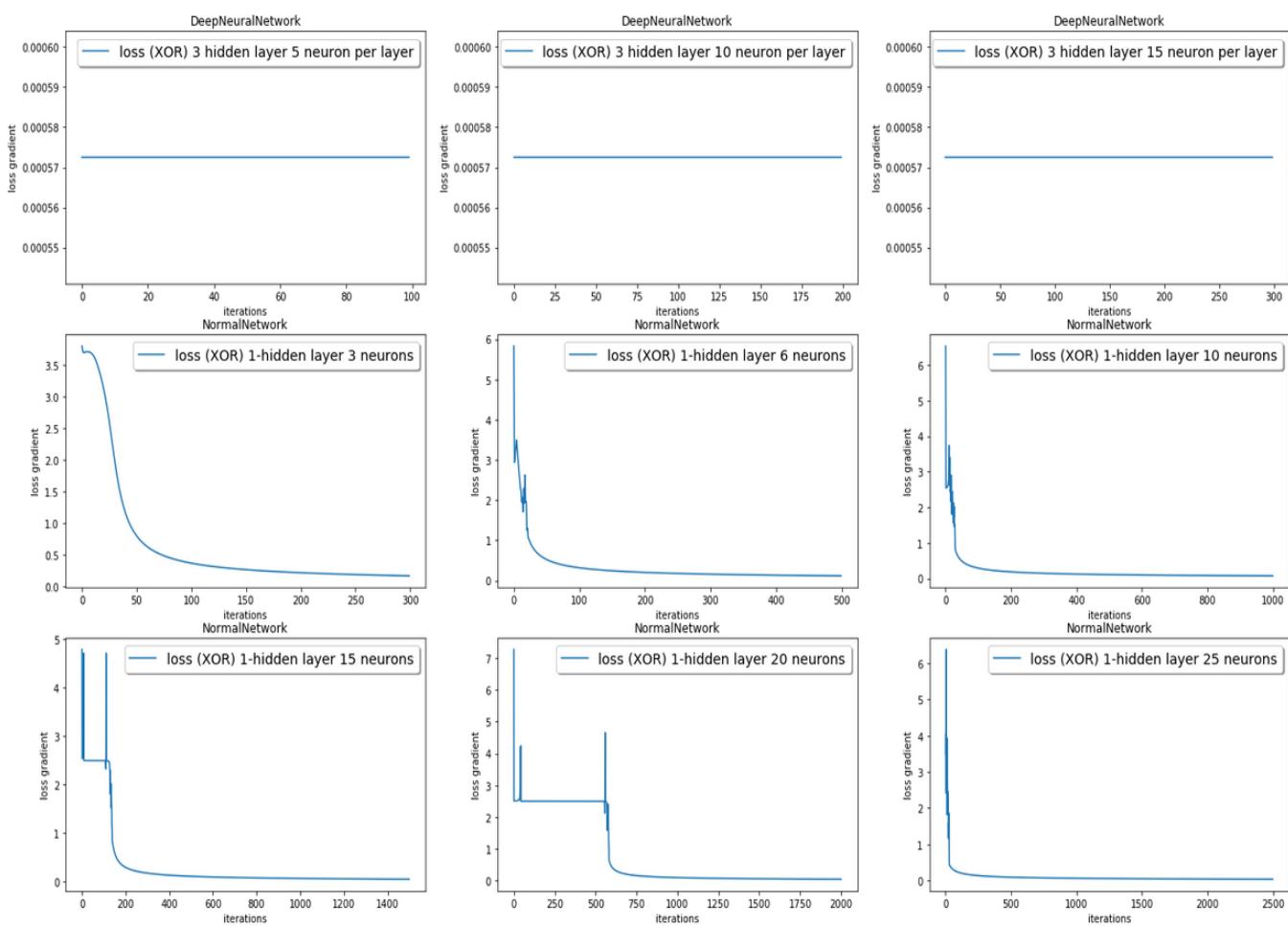
    if self.layersNum == 2:
        self.inputWeights = nn.Linear(self.inputSize, self.hiddenSize) # i X h tensor in to hidden 1
        self.fromFirstLayerTOutWeights = nn.Linear(self.hiddenSize, self.outputSize) # h X o tensor hidden 1 to out

    if self.layersNum == 3:
        self.inputWeights = nn.Linear(self.inputSize, self.hiddenSize) # i X h tensor in to hidden 1
        self.fromFirstLayerTSecondLayerWeights = nn.Linear(self.hiddenSize, self.hiddenSize) # h X h tensor hidden 1 to hidden 2
        self.fromSecondLayerTOutWeights = nn.Linear(self.hiddenSize, self.outputSize) # h X o tensor hidden 2 to out
```

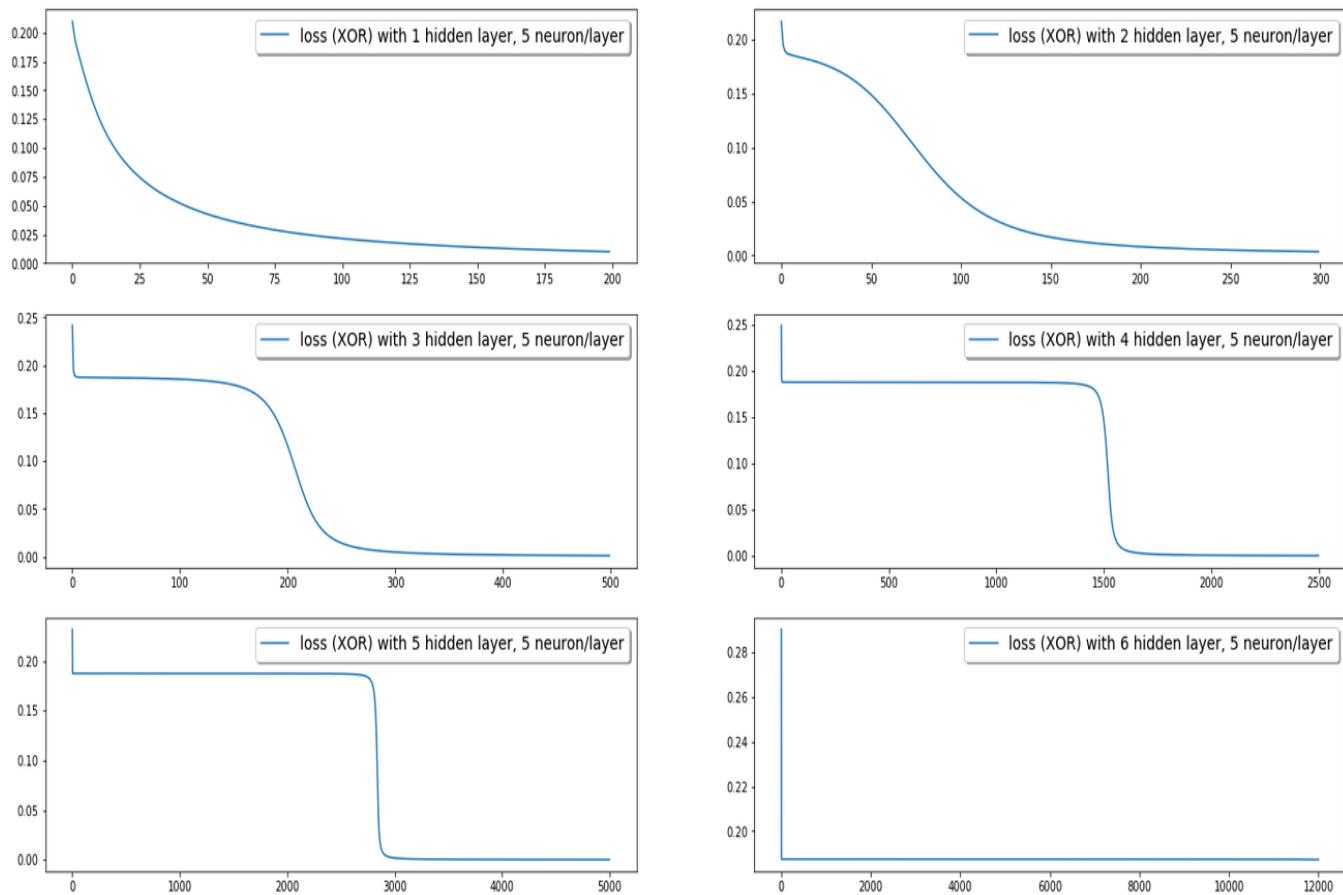
در نمودار زیر داریم:

ابتدا به کمک شبکه‌ی عمیق تعریف شده داریم که با تعداد 3 لایه شبکه را طراحی کرده و سپس تعداد نورون‌ها را 5 و 10 و 15 در نظر می‌گیریم و هم‌چنین در ادامه نمودار‌های حالت یک لایه مخفی وسط با تعداد نورون‌های 3 و 6 و 10 و 15 و 20 و 25 نیز کشیده‌ام

البته ذکر یک نکته مهم است: و آن این است که در ابتدا پس از چند بار train کردن شبکه داشتیم که به صورت نمایی سریعاً در حالت عمیق همگرا به صفر می‌گشت اما پس از چند بار train کردن در زمان‌های دیگر (یه زمان دیگر که لپتاپ رو روشن کردم!) مقدار loss به صورت زیر در حالت عمیق ایست کرد و با تکرار مکرر و ران کردن مکرر تغییر نکرد و این چنین ایست کرد.



بخش 7 قسمت دوم و بخش 8:



نتیجه گیری ها:

با افزایش تعداد لایه ها مشاهده می شود که سیر یادگیری کند تر می گردد و این به این تعبیر است که انگار مدتی طولانی در یک نقطه ای بحرانی گیر می کنیم و پس از آنکه همه لایه ها وارد این نقطه ای بحرانی شدند به سرعت مقدار خطا از این نقطه ای بحرانی خارج می گردد و سریعاً به صفر همگرا می گردد. همانطور که مشاهده می کنید تا نقطه ای مقدار loss تقریباً ثابت ادامه می یابد و ناگهان به سرعت به صفر کاهش می یابد.

جمع بندی کلی دو بخش آخر: شبکه ای **تک لایه** و نورون **زیاد** بله **سریعتر** یادگیری می کند و شبکه با **چند لایه** به مقدار درست تر و **دقیق** تر نهایی **همگرا** می گردد.

سوال دوم:

• بخش اول

در ابتدا تابعی نوشتیم که یک ماتریس باینری رندوم می سازد و خروجی بیت پریتی آن را نیز تا j امین بیت می دهد.

```
def binarySampleVsParity(row,column):  
  
    binary = np.zeros((row,column),dtype = float)  
    parity = np.zeros((row,column),dtype = float)  
  
    for j in range(column):  
        for i in range(row):  
  
            random = np.random.randint(0,2)  
            binary[i][j] = random  
  
            if(j == 0):  
                parity[i][j] = binary[i][j]  
            elif( parity[i][j-1] == 0 and binary[i][j] == 1 ):  
                parity[i][j] = 1  
            elif( parity[i][j-1] == 0 and binary[i][j] == 0 ):  
                parity[i][j] = 0  
            elif( parity[i][j-1] == 1 and binary[i][j] == 0 ):  
                parity[i][j] = 1  
            elif( parity[i][j-1] == 1 and binary[i][j] == 1 ):  
                parity[i][j] = 0  
  
    # binary = torch.from_numpy(binary)  
    # parity = torch.from_numpy(parity)  
  
    # binary = tf.tuple(binary)  
    # parity = tf.tuple(parity)
```

• بخش دوم: موجود در فایل پایتون زیپ ارایه شده.

• بخش سوم:

تقریبا شبکه دیگر کار نمی کند. علت این موضوع در آن است که ما طبق گفته‌ی سوال رشته‌های باینری به طول 10 تولید می کنیم و به شبکه می دهیم لذا شبکه کلا می تواند (به علت تولید رندوم داده‌ها) 1024 نوع (2 به توان 10) نوع رشته‌ی 10 تایی را یاد بگیرد. لذا شبکه ما برای داده‌های با طول نسبتاً نزدیک به 10 درست کار می کند اما برای داده‌های به طول زیاد مثلاً 100 دیگر شبکه نمی تواند درست کار کند. چرا؟

چون همانطورکه می دانید شبکه با داده‌های باینری به طول 10 آموزش دیده شده است در حالی که ما به آن رشته‌ی 100 تایی می دهیم. توجه کنید که رشته‌ی 100 تایی 2 به توان 100 حالت متفاوت دارد و این مقدار تقریباً برابر ده میلیارد نوع رشته می باشد حال آنکه ما فقط هزار تا رشته به شبکه آموزش داده ایم لذا طبیعی است که شبکه نتواند پاسخ مطلوب برای این سایز رشته بدهد.

هم چنین ایراد دیگری که وجود دارد این است که در اثر آموزش 1024 نوع رشته‌ی 10 تایی به نوعی شبکه over fit می شود و خب طبیعی هم هست چرا؟ خب آمده ایم فقط رشته‌های 10 تایی رو مدام به شبکه یاد می دهیم لذا طبیعی

است که شبکه حساس به این سایز و بعد باشع لنزا با دادن ورودی های با سایز نسبتاً متفاوت داریم که از پاسخ اصلی diverge می کند شدید!

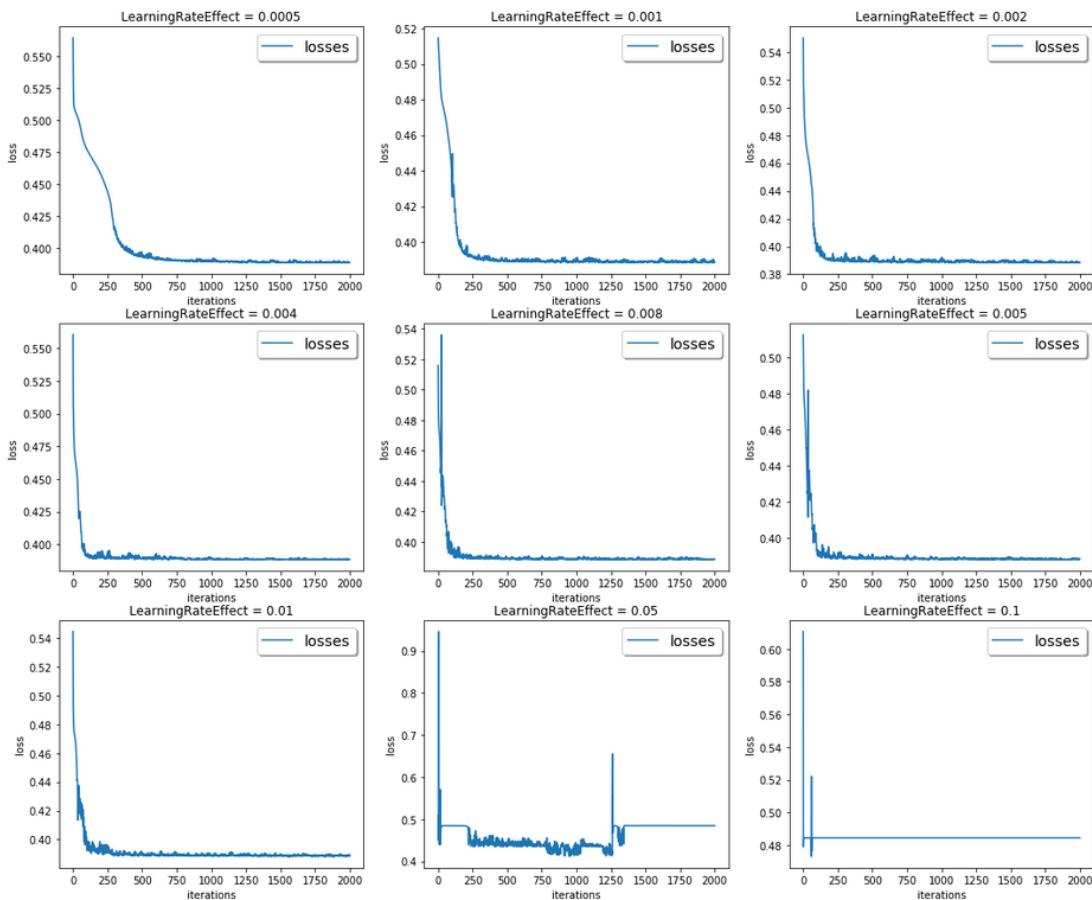
- نکته دیگر: البته همانطور که در نمودار های سوال اول و سوم و چهارم می بینید با افزایش تعداد بار های آموزش شبکه به نسبت می توان وضعیت یادگیری شبکه را بهبود بخشید لذا داریم که با افزایش مراتب تکرار یادگیری (حدودا 10.000 البته با ترفند هایی می توانستیم به 3000 هم برسیم) این موضوع بین همه ی گروه ها مشترک بود!!!) این مشکل تا حدی حل شد اما خیلی کند شد!

- نکته ی دیگر اینکه سختی تعلیم این تیپ شبکه ها به علت حافظه دار بودن بیش از حد سیستم می باشد همانطور که می بینید بیت پریتی به همه ی حافظه های قبلی نیاز دارد که این خود مشکل مارا زیاد می کند!

بخش چهارم:

- Graphs

- Learning Rate Effect:



نتیجه گیری: مشخص است که نرخ یادگیری هر چقدر زیاد باشد داریم که همگرایی به مقدار نهایی زودتر رخ می دهد و این خود به دلیل بیشتر شدن تعداد گام های برداشته شده در هر مرحله و اندازه ی گام ها می باشد. و توجه کنید که در هر

چقدر مقدار نرخ یادگیری کوچک باشد داریم که خطای زیاد تر می شود چرا که نمی تواند خطای زیر حد نرخ یادگیری را تغییر دهد.

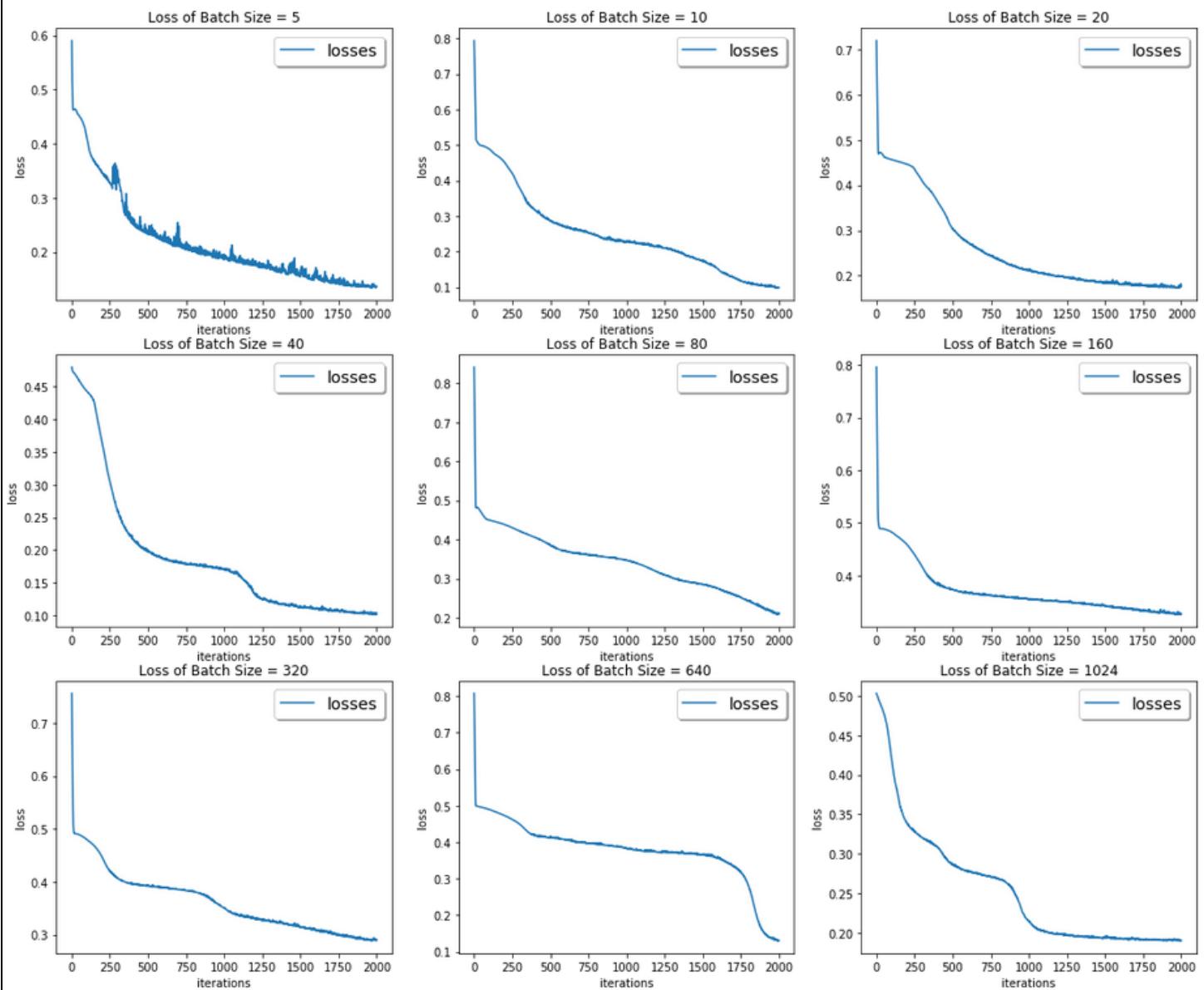
البته گاهی وقتاً اگر بیش از حد نرخ یادگیری را تغییر بدھیم و آگرا می شود!!!

که در اینجا دیگر چنین حالتی را نیاوردم ولی در آزمایش ها چنین حالت هایی نیز رخ داد. (مثال سوال اول)!

- Graphs...

- ...

- Batch Size



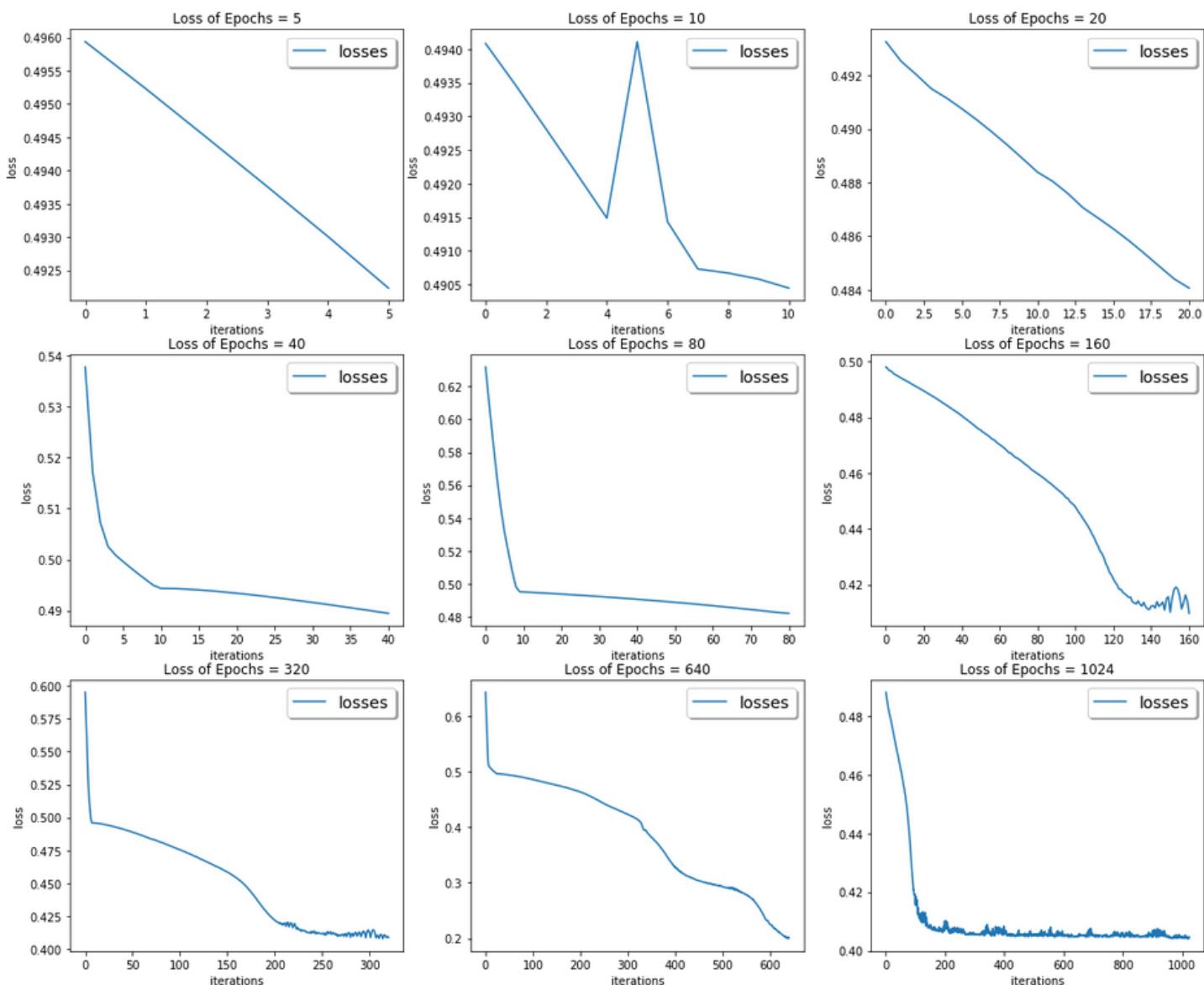
نتیجه گیری: در فرآیند یادگیری هر بار بسته‌ی داده‌ها به شبکه داده می‌شود و هر چقدر این بسته‌ی داده‌ها بزرگ باشند داریم که سیر خطای زودتر همگرا به صفر می‌گردد. (چراکه وقتی سایز بسته‌ها بیشتر باشد میزان داده‌های در کنار هم برای ترین شبکه افزایش یافته و شبکه بهتر یاد می‌گیرد و خطای حاصل کاهش می‌یابد...)

- Graphs:

- ...

- ...

- Number of epochs

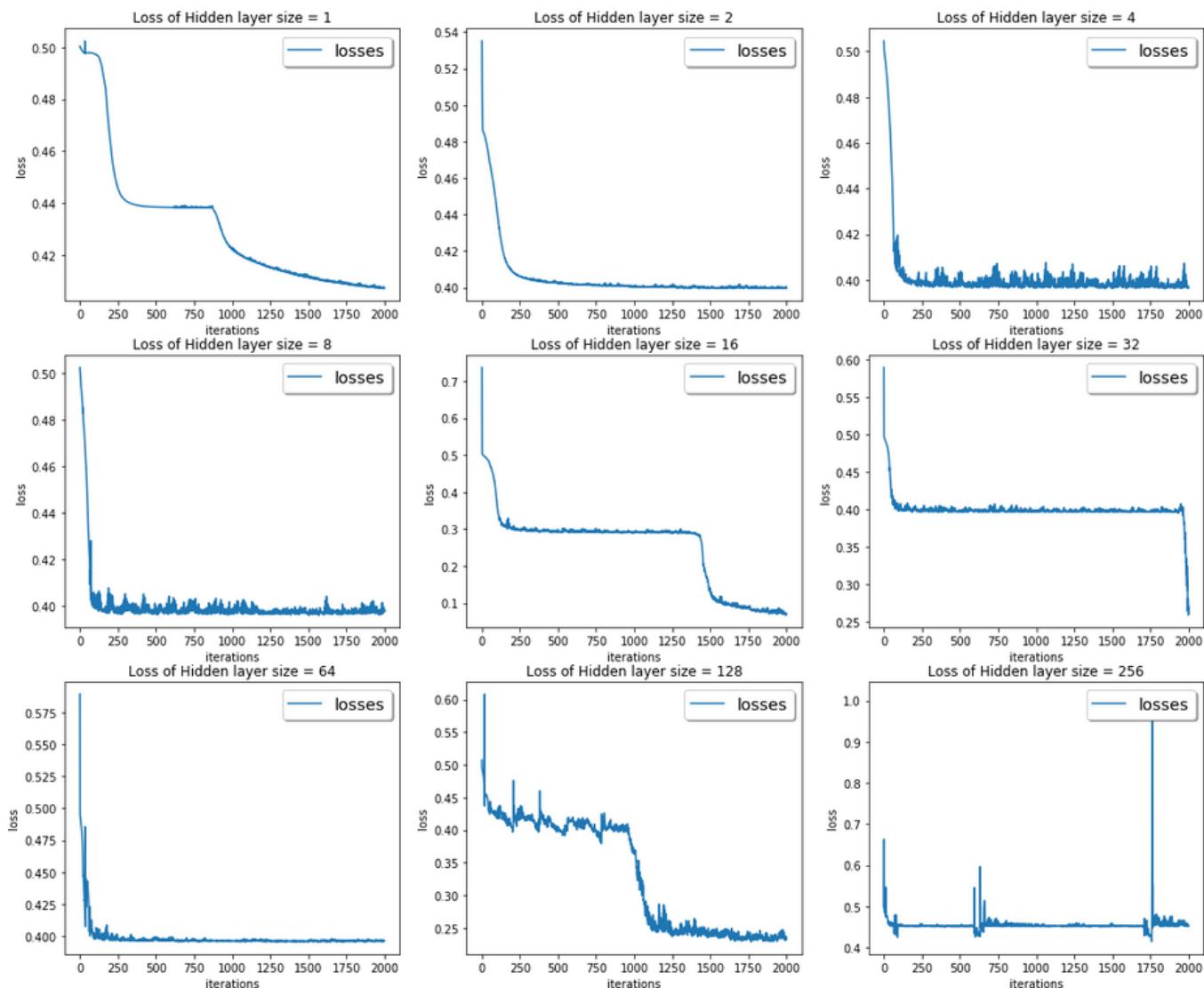


نتیجه گیری:

در فرآیند یادگیری داریم که با افزایش تعداد epoch ها میزان و سیر همگرا شدن loss به صفر بیشتر می گردد و این پارامتر یک ویژگی متمایز با سایر پارامتر های شبکه عصبی دارد و آن این است که این پارامتر هر چقدر زیاد رتر شود داریم که کمک بیشتری به همگرا شدن شبکه می کند و بدون اینکه زیادتر شدن بیش از حد آن برخلاف مثلا (Rate or number of hidden layers ...) باعث واگرایی شبکه شود و این خود امتیاز مشتبی است که این ویژگی شبکه دارد.

- Graphs:

- ...
- ...
- ...hiddenLayerSize



نتیجه گیری ها:

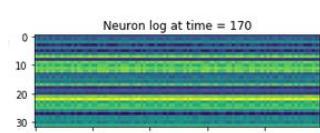
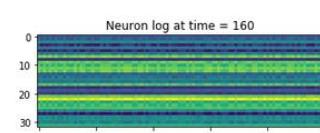
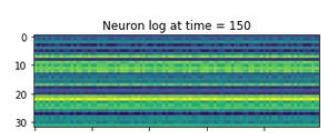
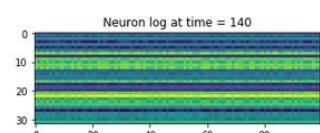
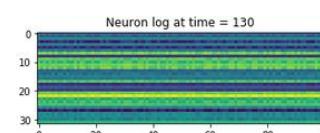
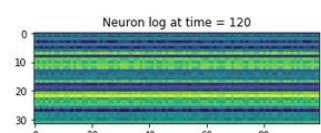
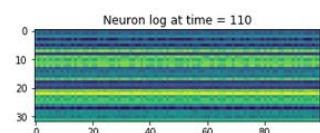
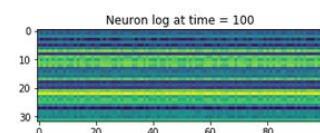
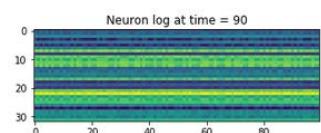
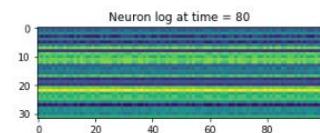
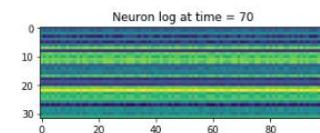
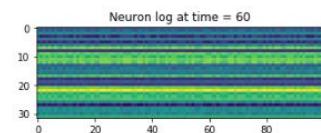
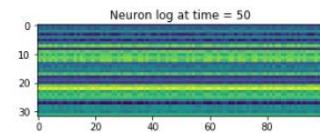
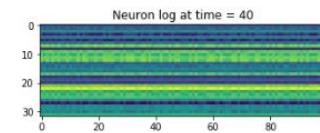
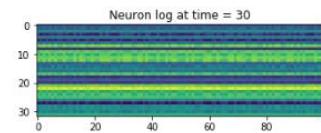
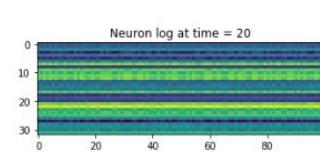
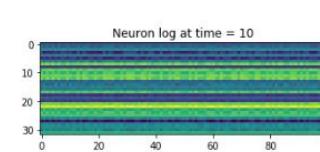
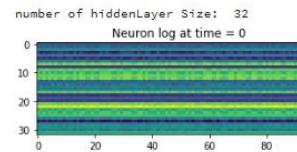
اتفاق عجیبی در این حالت رخ داده است و آن این است که در این حالت بر خلاف موارد مشابه که افزایش بیش از حد hiddenlayerSize شبکه و اگرا می شد داریم که اثر افزایش سایز لایه i میانی بسیار مطلوب می باشد از آنجا که پیش تر اشاره کردیم داشتیم که علت عدم توانایی در پاسخ دهنده به رشته های با طول بیشتر تفاوت زیاد order های ممکن باینری رشته i ورودی نسبت به order حالت های ممکن باینری داده های ترینی لذا گفتیم که فضای ایجاد شده توسعه شبکه که (بسیار به سایز لایه i پنهان بستگی دارد) نمی تواند آنقدر بزرگ باشد تا از overfitting و سایر خطاهای ایجاد شده در اثر دادن ورودی جدید و کمتر نزدیک! جلو گیری کند اما در این نمودارها همانطور که می بینید (سایز لایه میانی به صورت نمایی افزایش یافته است) مقدار loss و خطای در اثر افزایش سایز لایه i میانی (به تبع آن بیشتر شدن و بزرگ تر شدن فضای تولیدی شبکه ...) کاهش می یابد.

توجه اگر خیلی افراط کنیم و خیلی سایز لایه i پنهان را زیاد کنیم ممکن است diverge رخ بدهد و آن زمانی است که سایز و فضای شبکه از threshold لازم برای یادگیری و ترین کامل شبکه بیشتر شود ...

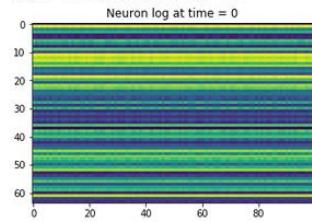
بخش 5: فعالیت نورون ها

در این قسمت به دلخواه داده های قسمت اول بخش قبل را در نظر گرفتیم و فعالیت نورون های آن قسمت را ترسیم کردیم. توجه کنید من برای همه i learning Rate ها این کار را کردم و learning Rate مربوطه در کنار نمودار گفته شده است.

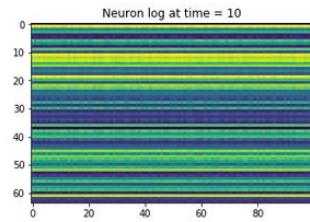
- نتیجه گیری: توجه کنید که نمودار های صفحه i بعد نشان می دهید که چقدر فعالیت نورون ها به مقادیر وزن ها در ابتدا بستگی دارد و داریم که در اصل می دانیم که تعداد نورون ها زیاد می باشد و داریم که عملکرد و فعالیت شبکه در نهایت متنها به همین فعالیت های نورون ها دارد و لذا داریم که تغییرات در وزن ها باعث تغییرات در فعالیت نورون ها می گردد و افزایش یافتن یک وزن متنها به فعال تر شدن فعالیت نورون مرتبط با آن وزن می گردد حال آنکه کمتر شدن یک وزن حین آموزش منجر به کاهش فعالیت نورون مربوط به آن وزن می گردد و داریم که این تغییر فعالیت ها به کمک رنگ ها و transient رنگ ها در نمودار های صفحه i بعد مشخص تر است.



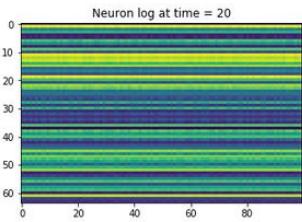
number of hiddenLayer Size: 64



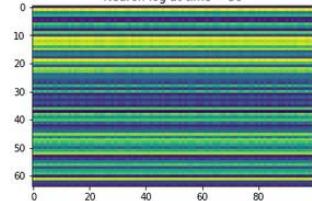
Neuron log at time = 10



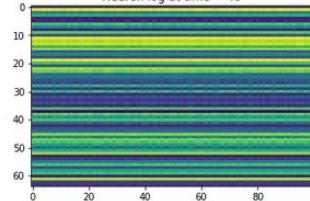
Neuron log at time = 20



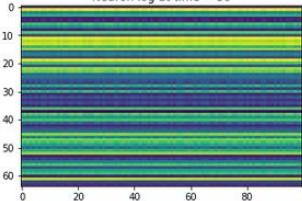
Neuron log at time = 30



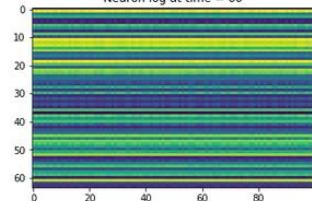
Neuron log at time = 40



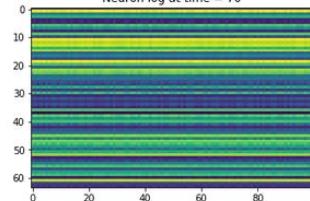
Neuron log at time = 50



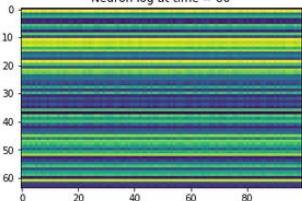
Neuron log at time = 60



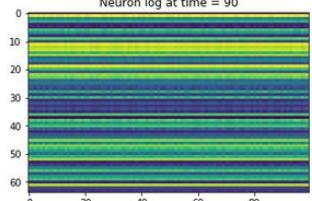
Neuron log at time = 70



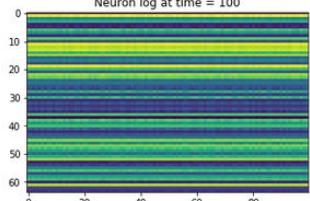
Neuron log at time = 80



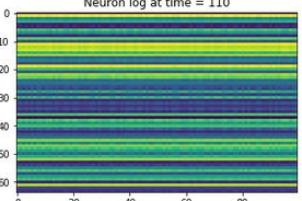
Neuron log at time = 90



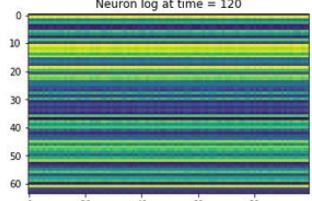
Neuron log at time = 100



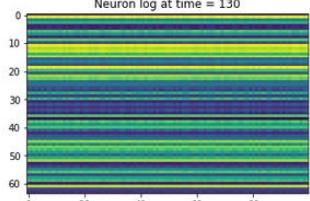
Neuron log at time = 110



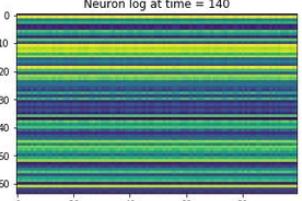
Neuron log at time = 120



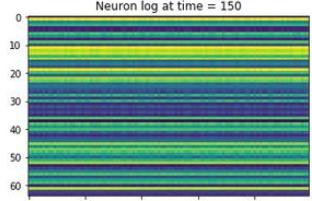
Neuron log at time = 130



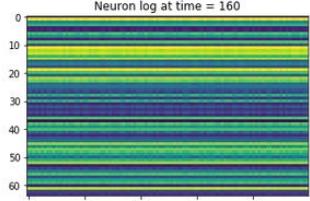
Neuron log at time = 140



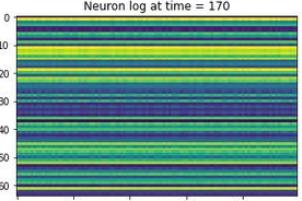
Neuron log at time = 150

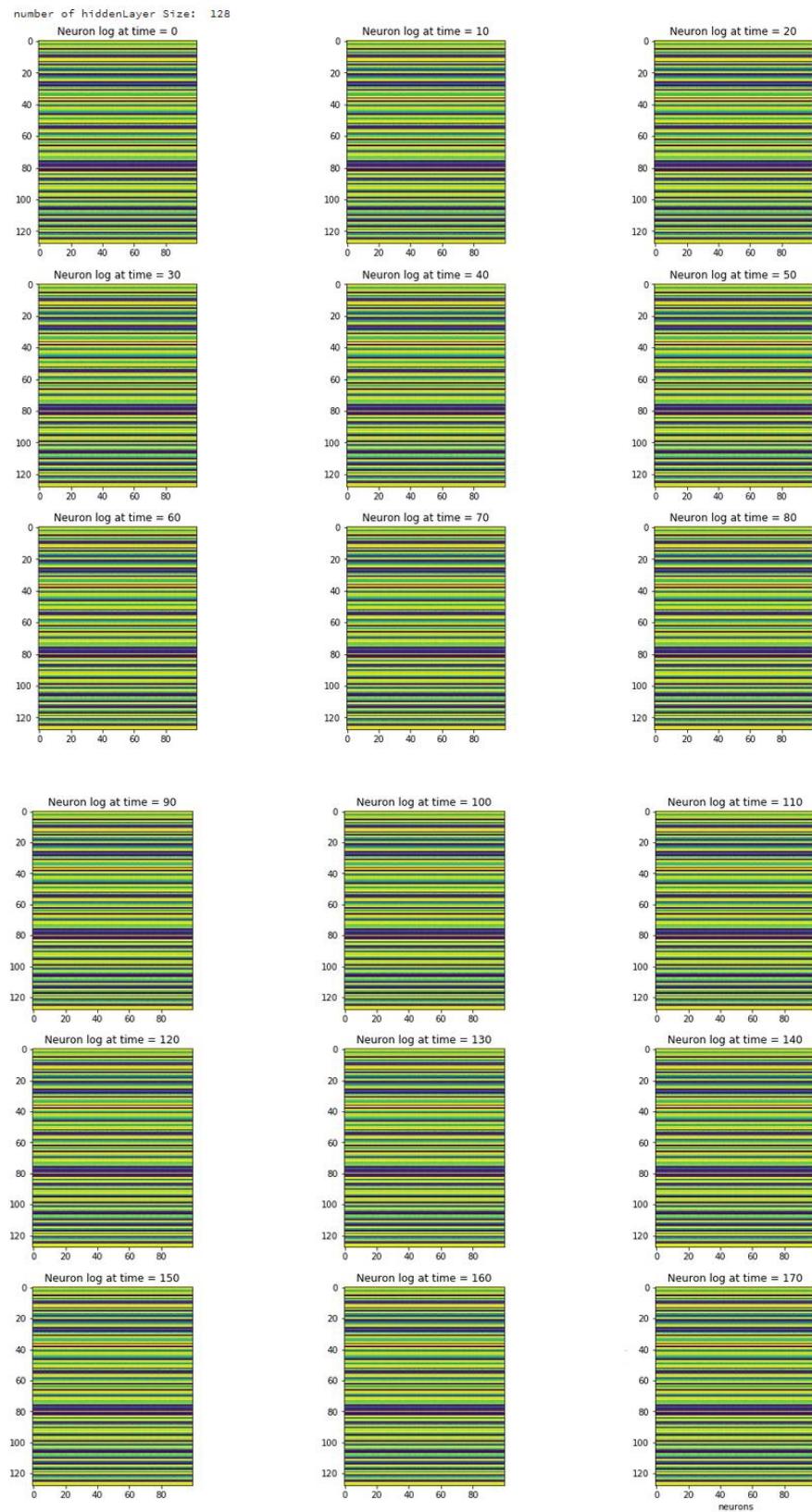


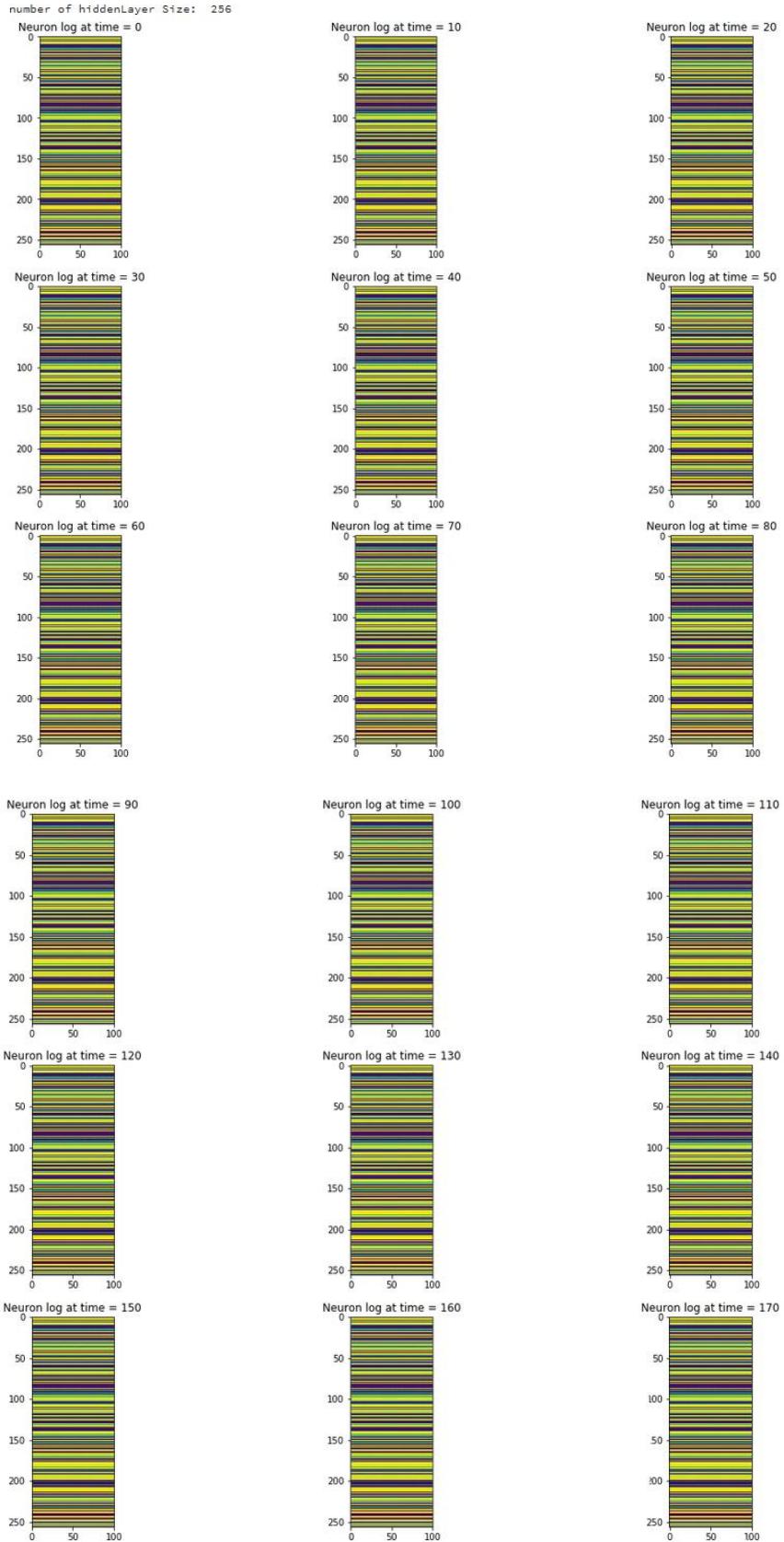
Neuron log at time = 160



Neuron log at time = 170







سوال سوم:

بخش های اول، دوم، سوم و چهارم:

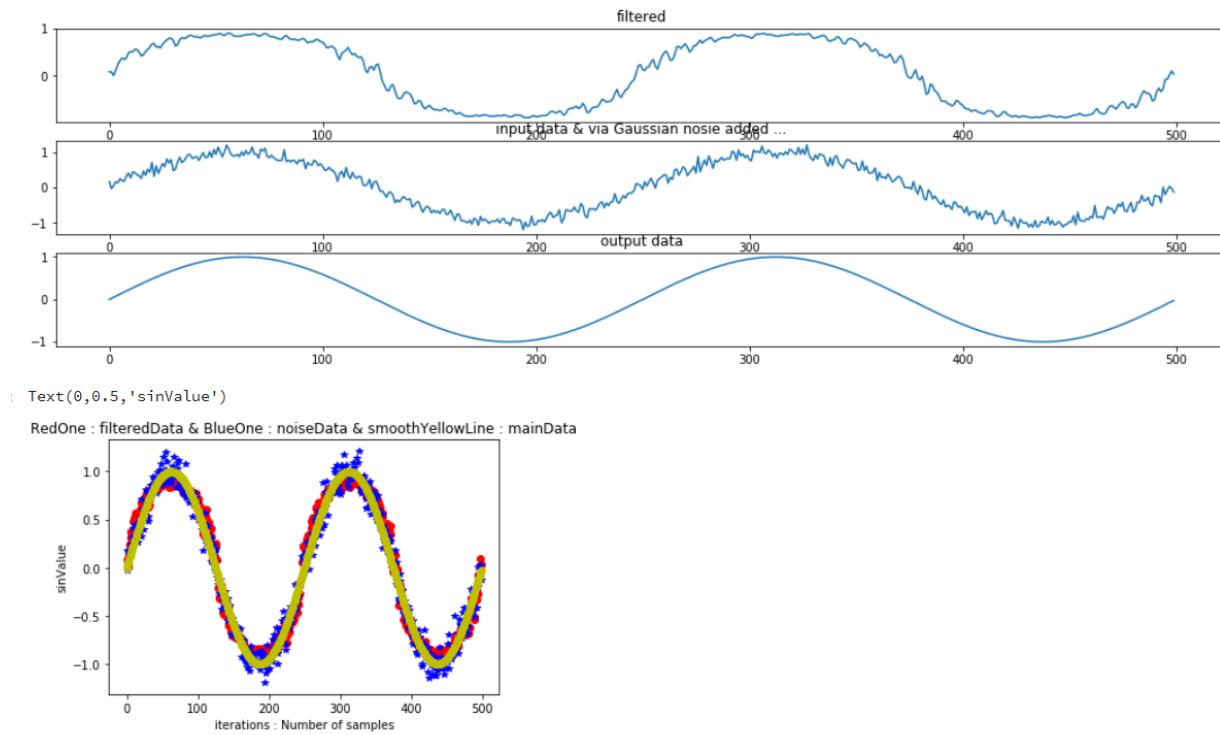
- نکته: توجه کنید لا یبری scripy.signal همه‌ی این سیگنال‌ها را دارد و به کمک signal این موج‌ها را تولید می‌کنیم البته می‌توانستیم توابع را منحصر به کمک مرج کردن تعداد ماتریس‌ها به دست بیاوریم که در برخی موارد این پیاده‌سازی را نیز آورده‌ام.

اما انتگرال‌گیری را به کمک قسمت علامت زده شده در کد شبیه‌سازی کرده‌ام.

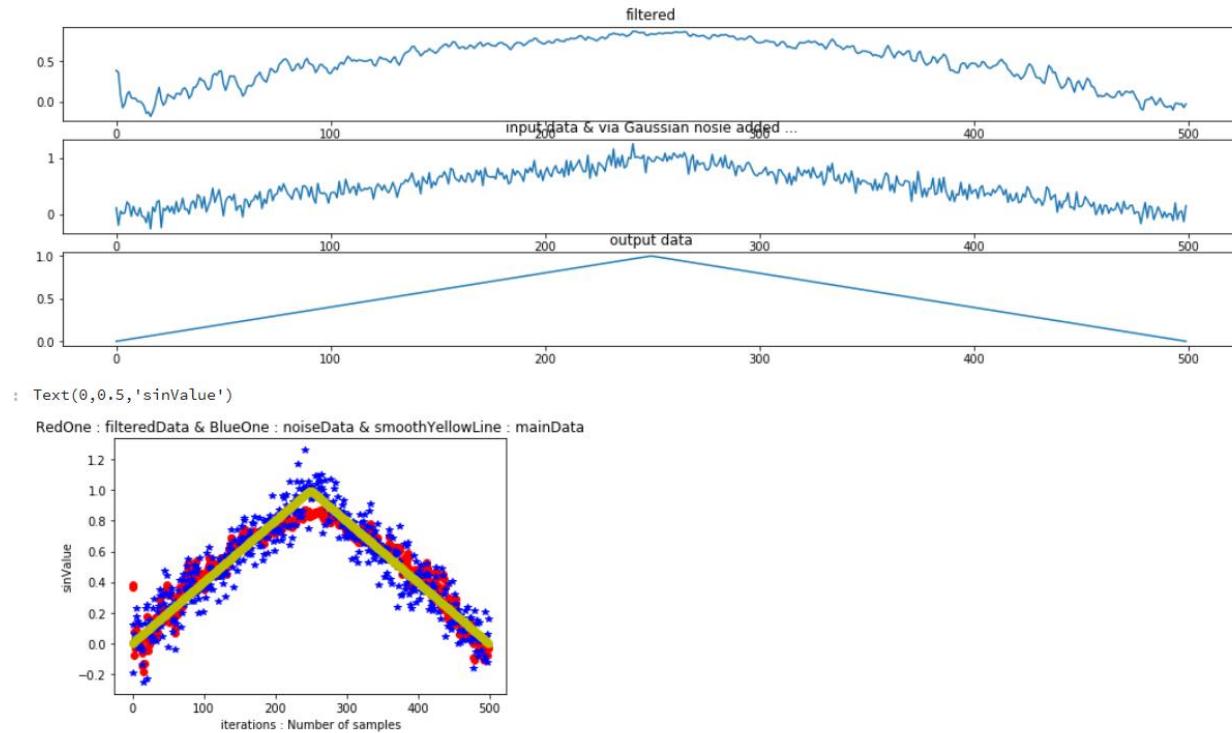
```
def sinWave(amp, signal_period, signal_size):  
    a = amp * np.sin(np.arange(signal_size) * (2 * np.pi / signal_period))  
    return a  
  
# Triangle wave  
def triangleWave(amp, signal_period, signal_size):  
    return signal.triang(signal_size)  
  
# Note: Another implementation ...  
#  
# we can also recreate triangle wave via creativity ...  
#  
#     u_period = np.linspace(-amplitude, amplitude, signal_period)  
#     l_period = np.linspace(amplitude, -amplitude, signal_period)  
#     a = np.concatenate([u_period[1:], l_period[1:]] * int(signal_size/signal_period), 0)  
#     return a[:signal_size]  
#  
  
# Teeth wave  
def teethWave(amplitude, signal_period, signal_size):  
    f_period = np.linspace(-amplitude, amplitude, signal_period)  
    a = np.concatenate([f_period] * int(2 * signal_size/signal_period), 0)  
    return a[:signal_size] #signal.sawtooth(2 * np.pi * 5 * signal_size)  
  
  
def noiseSignal(inputSignal_gonna_have_noise, mode, mean, var):  
  
    if mode == 'uniform':  
        noise = np.random.uniform(mean, var, size=inputSignal_gonna_have_noise.shape)  
    elif mode == 'normal':  
        noise = np.random.normal(mean, var, size=inputSignal_gonna_have_noise.shape)  
  
    # create new output Matrix with the same matrix size as input signal ...  
    brownian = np.zeros(inputSignal_gonna_have_noise.shape)  
  
    # causality!!!  
    brownian[0] = noise[0]  
  
    # finding the integral of noise ...  
    for i in range(1, int(inputSignal_gonna_have_noise.shape[0])):  
        # as we know brownian signal is integral of noise signal  
        # so as we read in discrete forms we know that  
        #  $y[n] - y[n-1]$  is equal to derivative of the signal  
        # we know that the brownian signal is integral of noise of signal so its derivative is equal to the noise signal  
        # so we have the equation :  
        #  $brownian[n] - brownian[n-1] = noise[n] \rightarrow brownian[n] = brownian[n-1] + noise[n]$   
        # we considered the initial point of brownian[n] as zero ... (causality ...)  
        # so U see the equation below ...  
        brownian[i] = noise[i] + brownian[i-1]  
    # return  
    return inputSignal_gonna_have_noise + noise, inputSignal_gonna_have_noise + brownian
```

بخش پنجم:

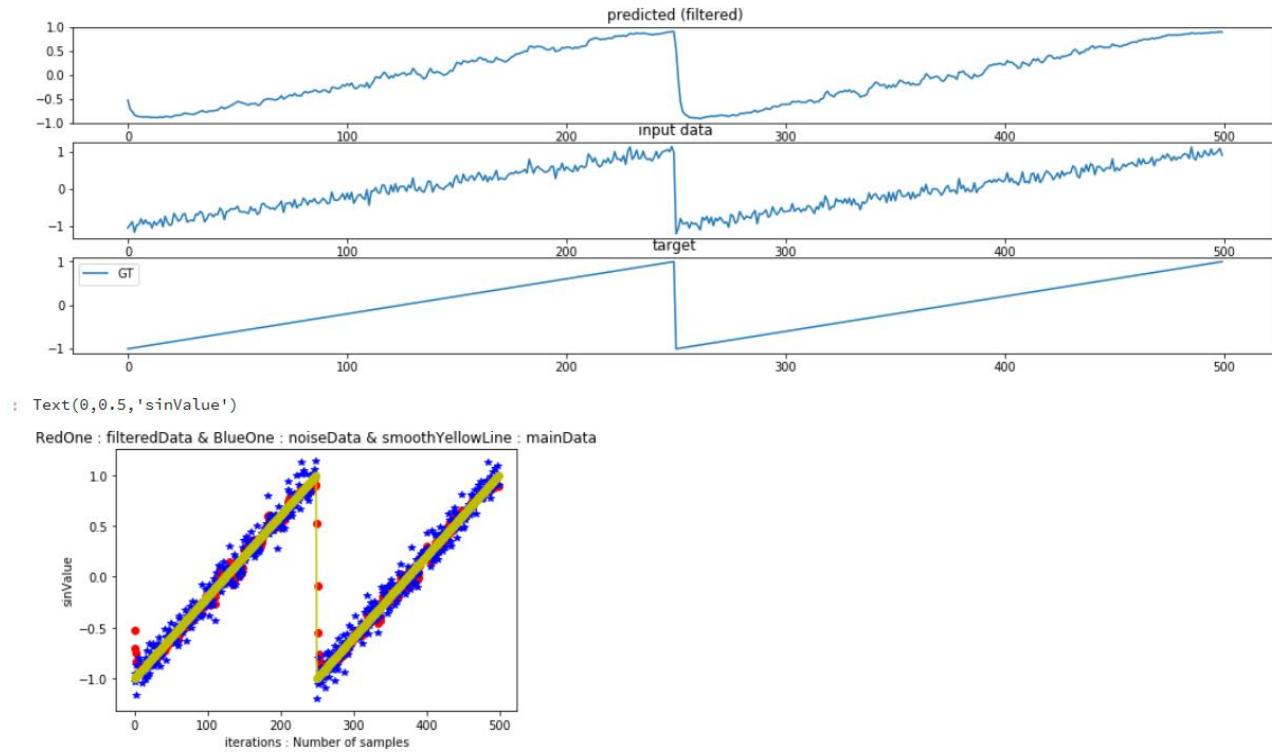
Sinus wave Signal via Gaussian Noise:



Triangle wave Signal via Gaussian Noise:

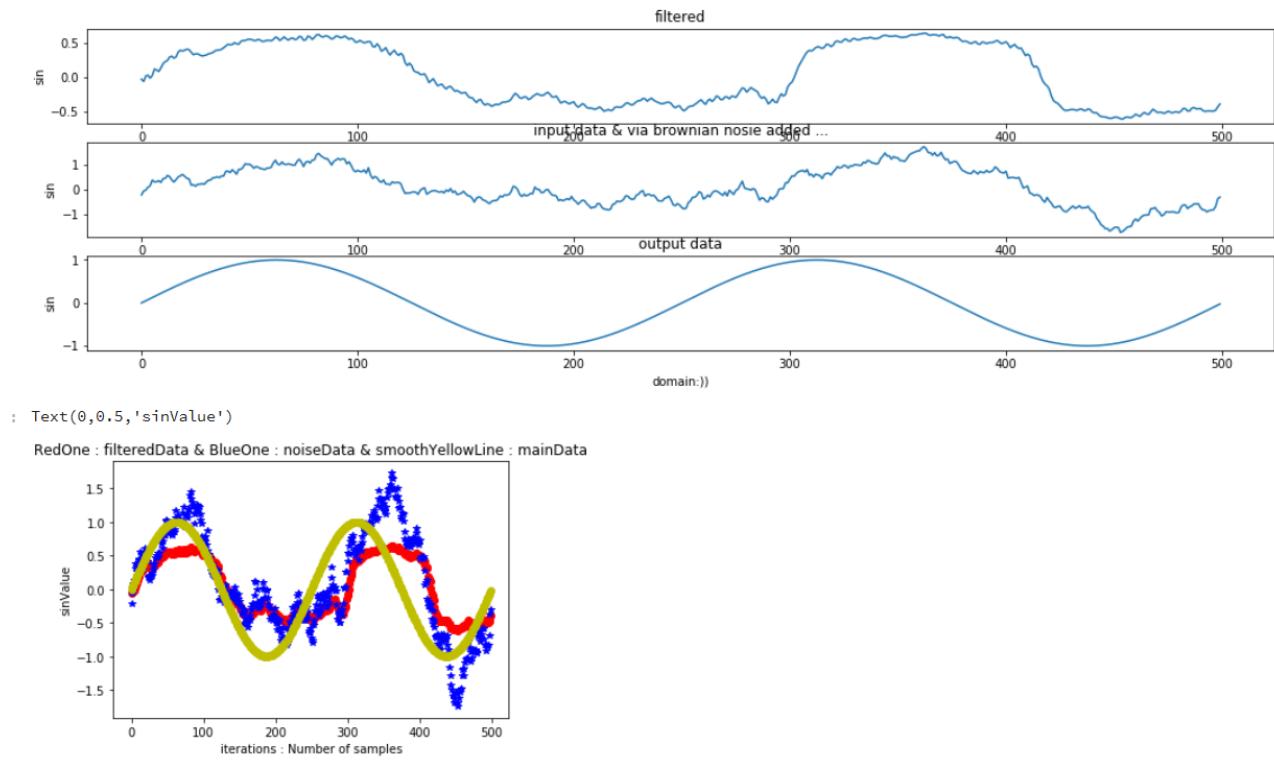


Teeth wave Signal via Gaussian noise:

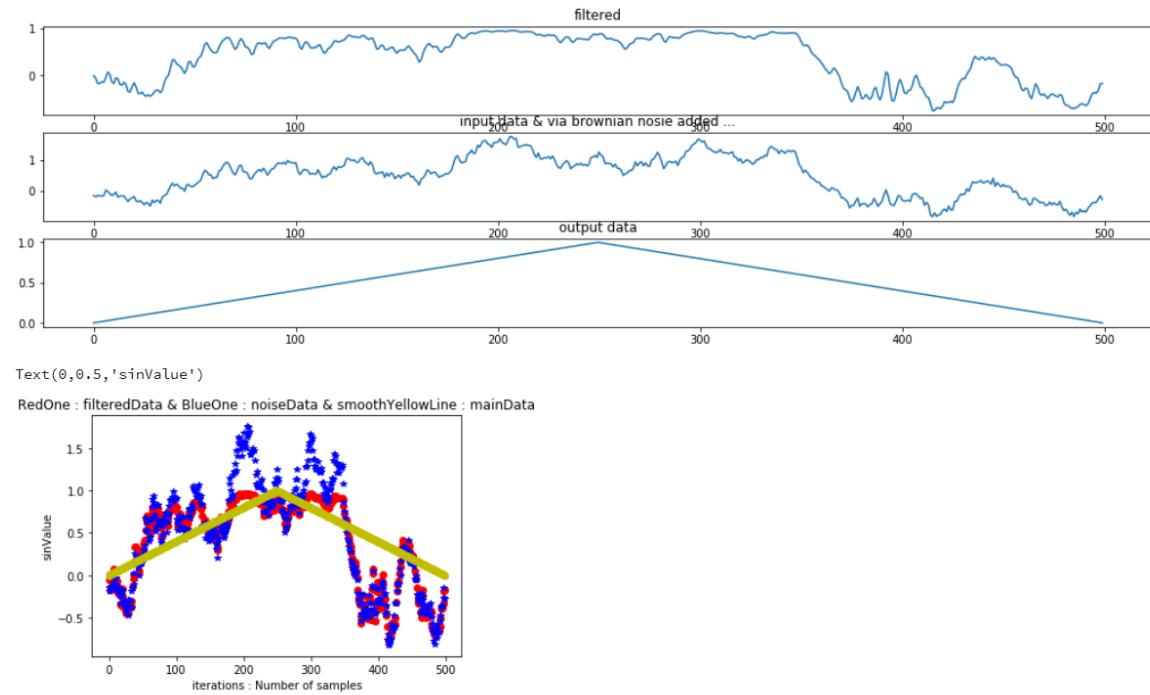


بخش ششم:

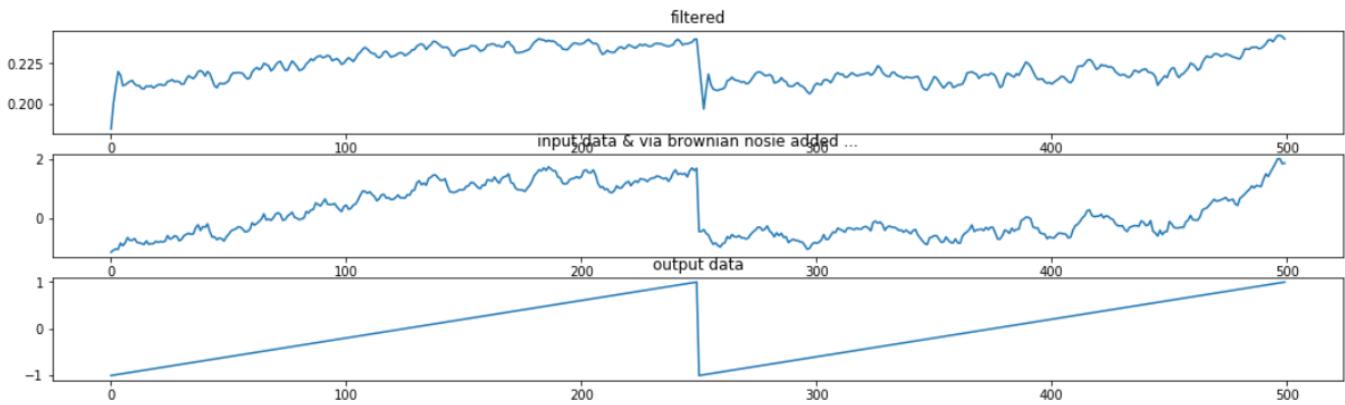
Sinus wave Signal via Brownian noise:



Triangle wave Signal via Brownian noise:

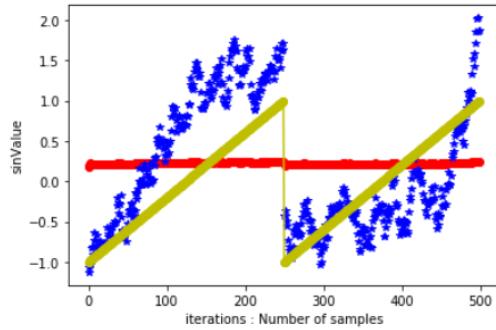


Teeth wave Signal via Brownian noise:



Text(0,0.5,'sinValue')

RedOne : filteredData & BlueOne : noiseData & smoothYellowLine : mainData



QuickOverview on SNR:

Signal to noise ratio (SNR) is a **local** relation between a signal $x[n]$ and noise $w[n]$ and is defined as:

$$\text{SNR} = \frac{\sigma_x^2}{\sigma_w^2}$$

where σ_x^2 and σ_w^2 denote the powers of the discrete-time signal $x[n]$ and noise $w[n]$ respectively.

Therefore if the signals involved are **non-stationary** of any sort, then the computed SNR will change from point to point as the signals change their character. One immediate application of this concept is in the audio industry known as those Dolby noise reduction systems, which roughly relies on the time varying SNR computation and SNR based gain control in the recording and playback systems to minimize hearable audio noise in the music.

For this reason, as a single parameter, SNR only makes sense iff the signals involved are **stationary** or at least WSS. Otherwise an average SNR can provide little help.

Then the theoretical computations of σ_w^2 and σ_x^2 depend on the signal models being employed. Since **noise** is by definition a random signal and we restrict it to a zero mean WSS noise process, then its theoretical power is computed in terms of an expectation;

$$\sigma_w^2 = E\{w(n)^2\}$$

for a zero mean random process, this expectation is equivalent to the variance of the noise process; i.e.,

$$E\{w[n]^2\} = E\{(w[n] - E\{w[n]\})^2\} = \text{Var}(w[n]) = \sigma_w^2$$

Note that in theoretical computation of noise power, it's more useful to consult to the auto-correlation sequence (ACS) of the WSS noise process as:

$$E\{w[n]^2\} = r_{ww}[0]$$

where ACF of a WSS process is defined as $r_{ww}[m] = E\{w[n]w[n+m]^*\}$ and which has a practical estimation based on ergodicity assumption below.

The signal $x[n]$ will have a deterministic definition and its average power can be computed as:

$$\sigma_x^2 = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2$$

by replacing $x[n]$ with the formula that defines it and by explicitly carrying out the summation along the signal length N for a closed form expression of the signal power. Note that even though a variance symbol is used to represent the average power of $x[n]$, it's a deterministic signal (at least in the theoretical computation)

In a **practical** setting, variance of the noise is computed by **estimating** it from a **time-average** of the given realization $w[n]$ in the time domain, relying on the assumption of **ergodicity** of the white noise process. Hence ;

$$\sigma_w^2 = E\{w[n]^2\} = r_{ww}[0] \longleftrightarrow \frac{1}{N} \sum_{n=0}^{N-1} w[n]w[n+0]^* = \frac{1}{N} \sum_{n=0}^{N-1} |w[n]|^2$$

Similarly in a practical setting the power for the deterministic signal is computed by the average sum of the sample squares:

$$\sigma_x^2 = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2$$

where N is the extent of the segment along which an SNR is computed.

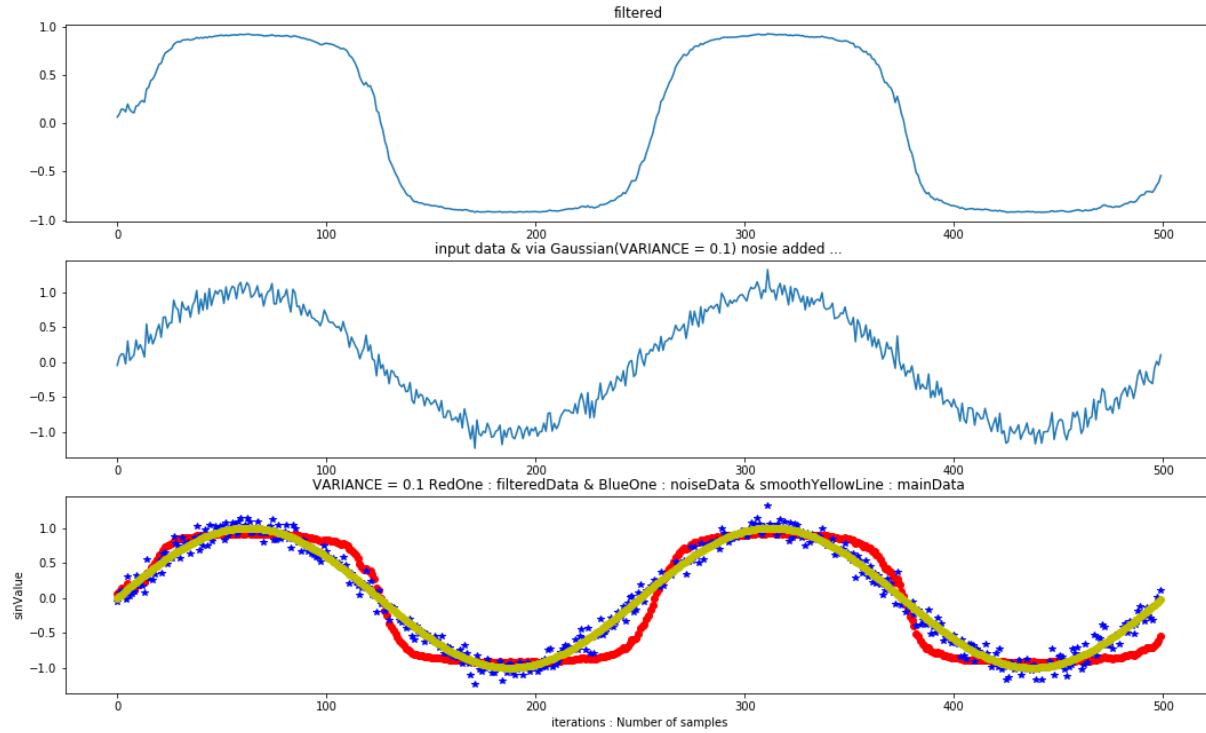
Note that even though the theoretical computation of the SNR (theoretical computations of the signal and noise powers) relied on different procedures used for the signal power and noise power, the practical computation of the SNR relies on exactly the same procedure for both powers.

Finally, in most cases SNR is described in a logarithmic scale with dB units;

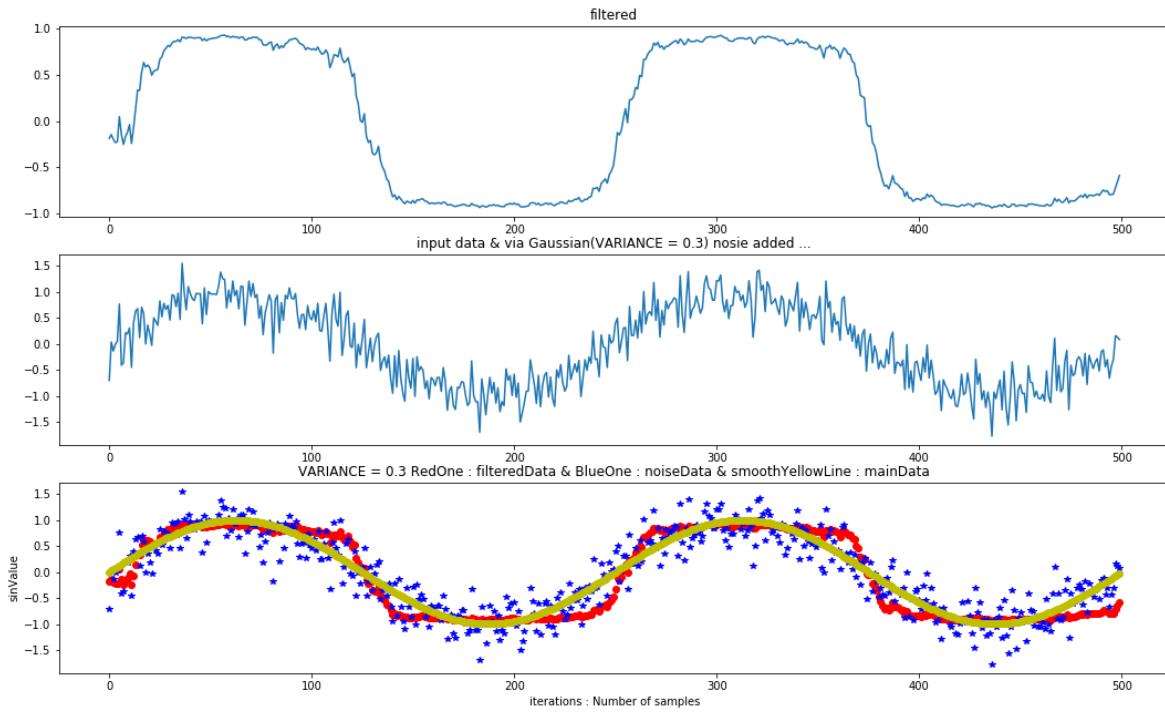
$$\text{SNR}_{dB} = 10 \log_{10}(SNR) = 10 \log_{10}\left(\frac{\sigma_x^2}{\sigma_w^2}\right)$$

بخش هفتم(تحلیل SNR): همانطور که ملاحظه می کنید با افزایش واریانس خطا SNR نیز افزایش می یابد.

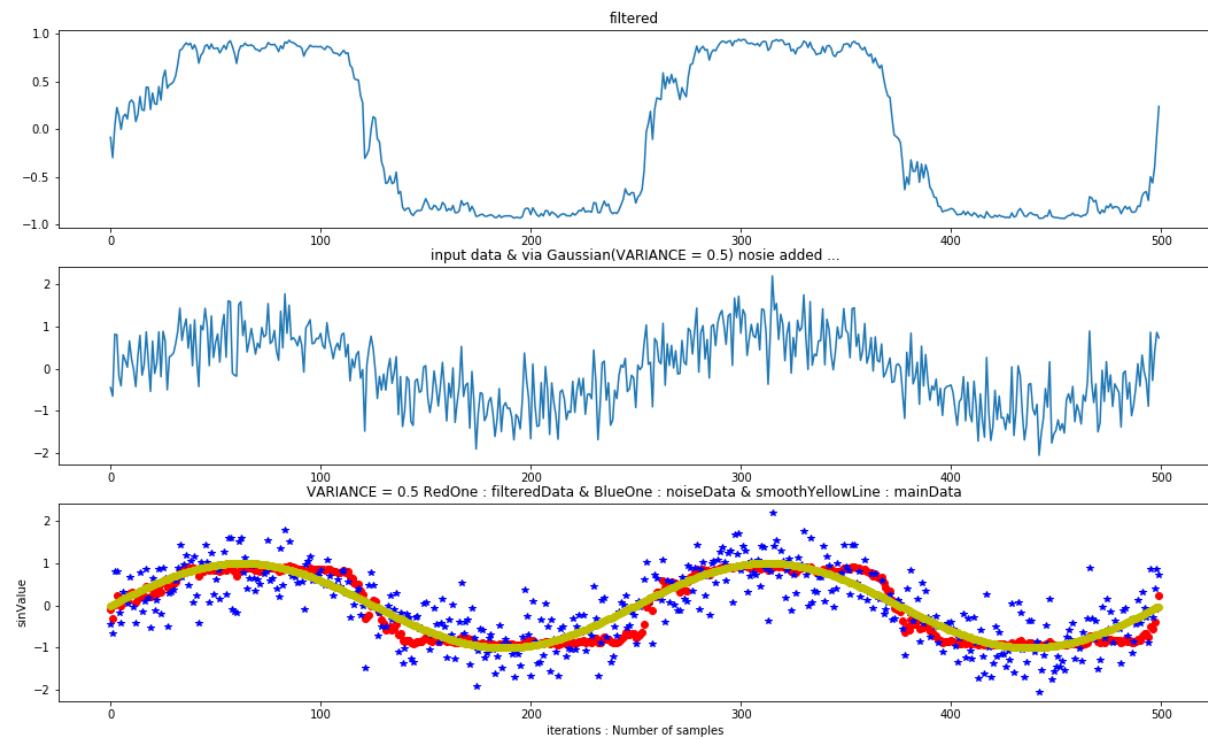
SNR is: 0.6456566759551717



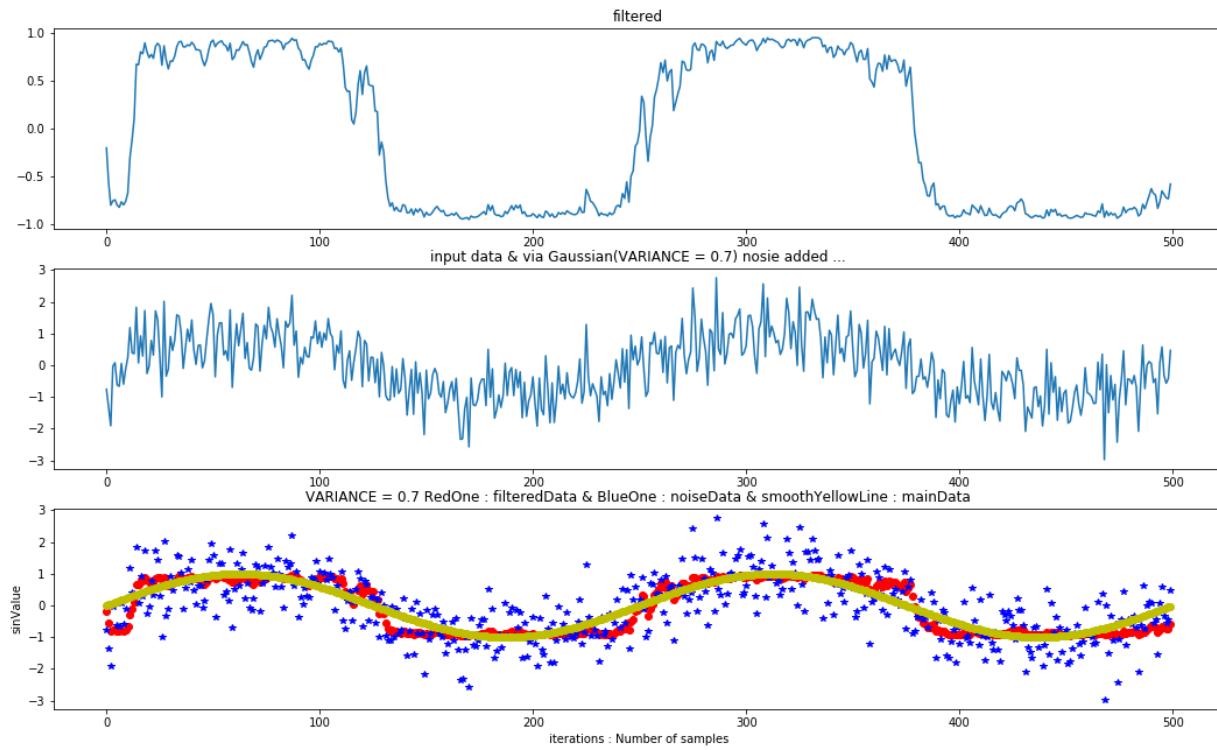
SNR is: 0.6696781068946754



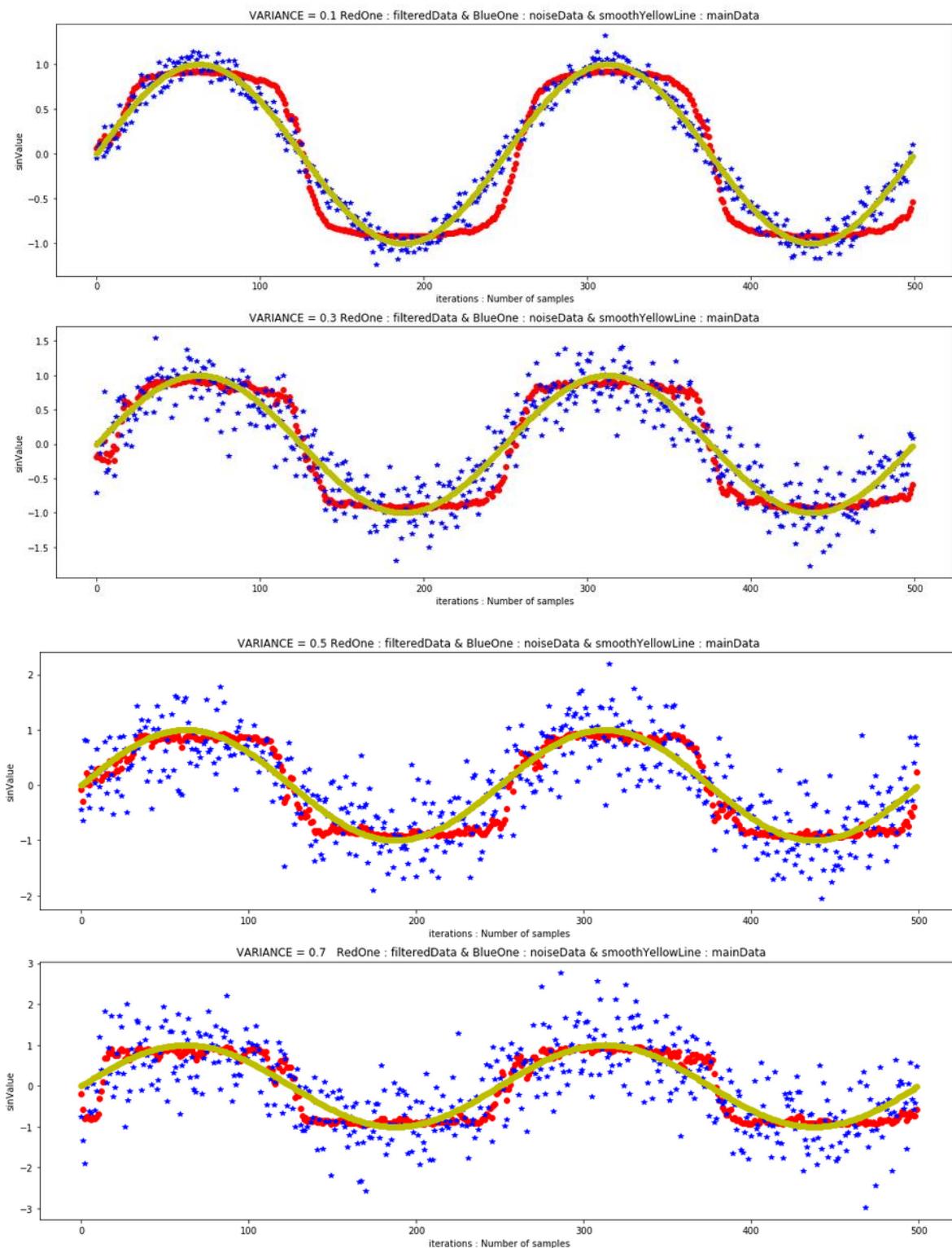
SNR is: 0.7161205638869154



SNR is: 0.8075311450260901



حال همه ی 4 حالت واریانس با هم دیگر در کنار هم: (برای آنکه حس بهتری پیدا کنیم)

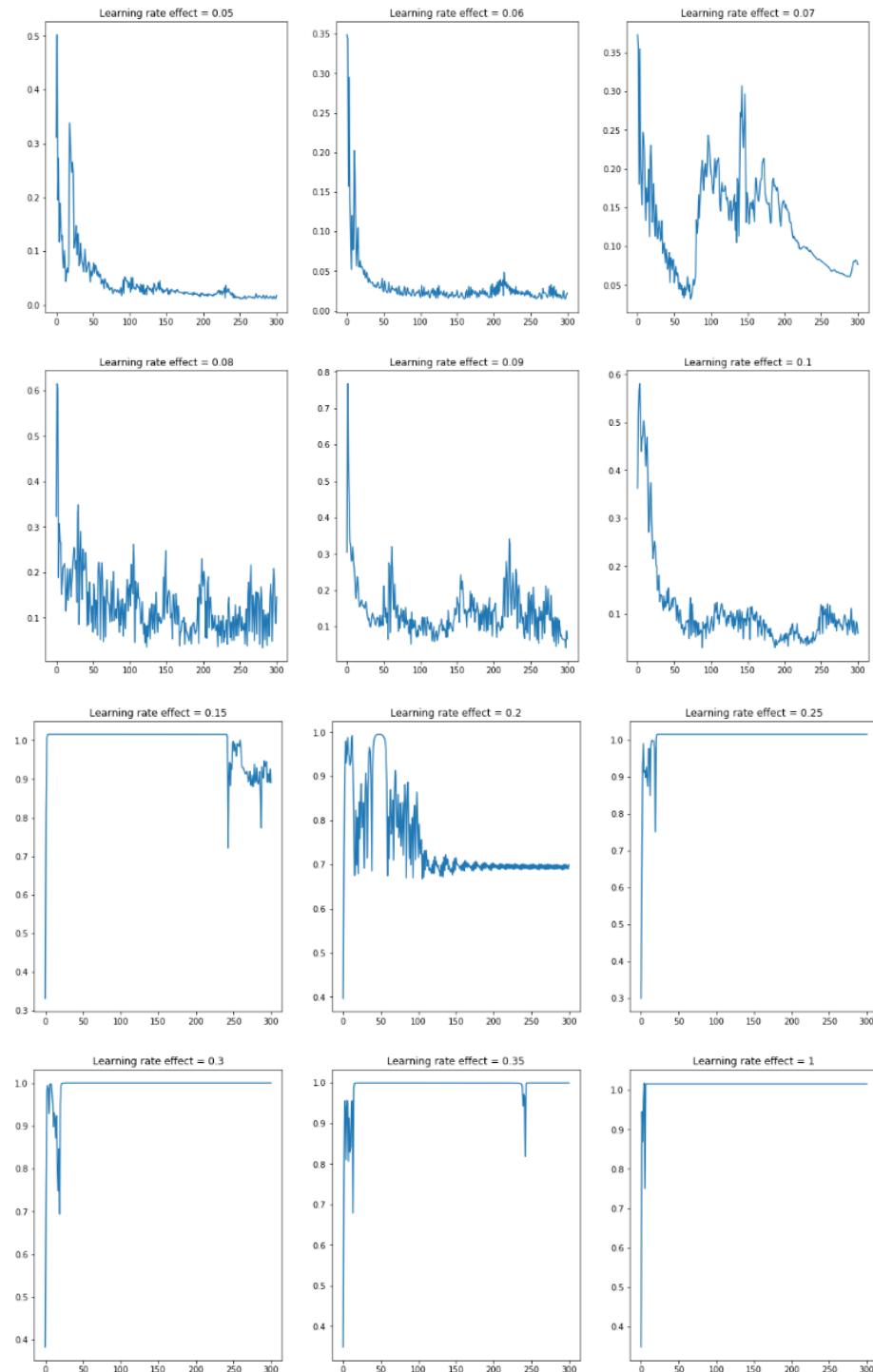


--- در این مساله پس از بررسی دیدیم که با افزایش واریانس داریم که مقدار SNR آن افزایش می یابد و همانطور

که ملاحظه می کنید با افزایش واریانس نویز اعمالی خطوط قرمز که فیلترینگ می باشند به خط زرد که نمودار اصلی می باشد نزدیک تر می شوند.

بخش 8:

قسمت اثر learning Rate •

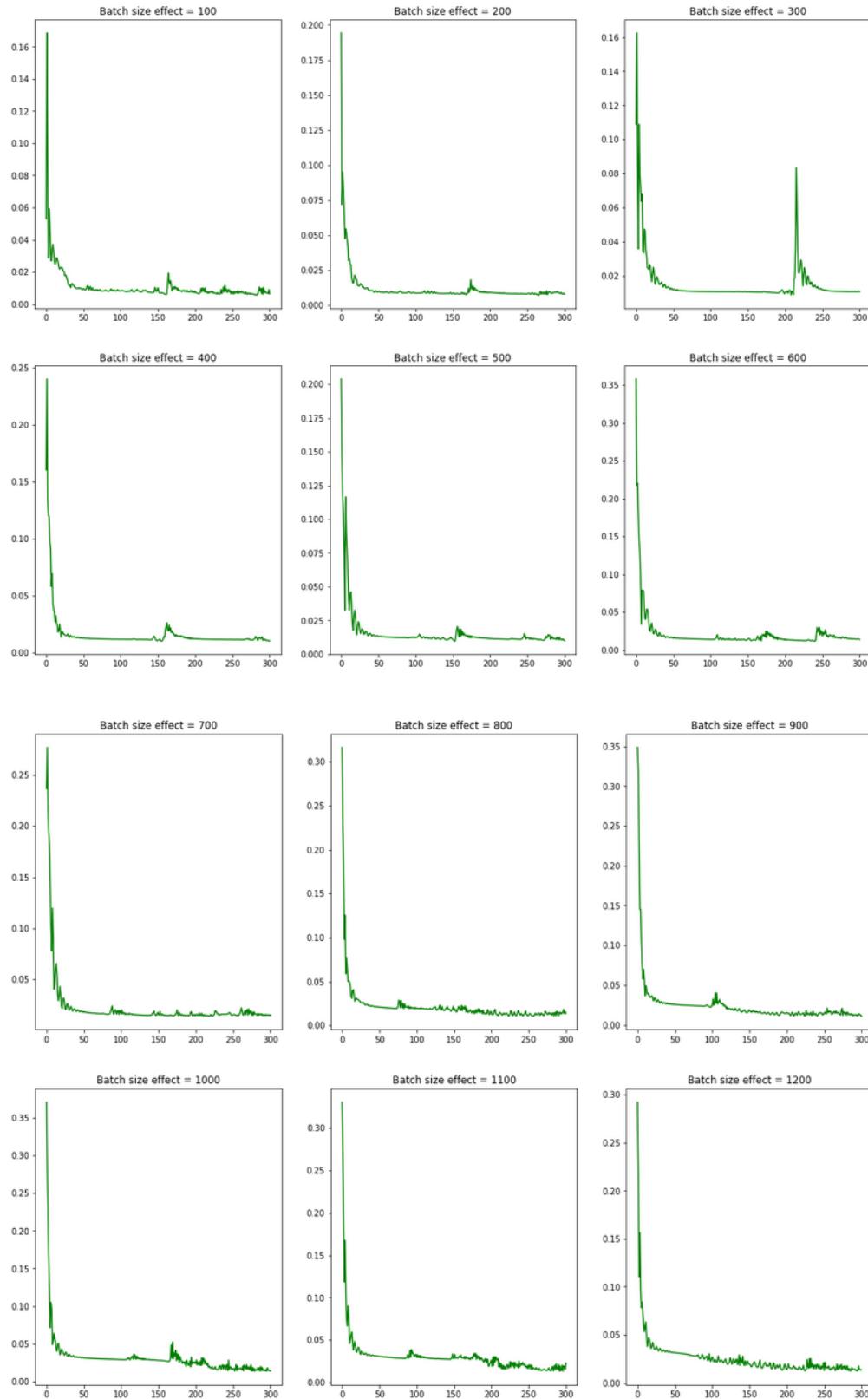


نتیجه گیری: مشخص است که نرخ یادگیری هر چقدر **زیاد** باشد داریم که همگرایی به مقدار نهایی **زودتر** رخ می دهد و این خود به دلیل بیشتر شدن تعداد گام های برداشته شده در هر مرحله و اندازه ی گام ها می باشد. و توجه کنید که در هر چقدر مقدار نرخ یادگیری کوچک باشد داریم که خطای زیاد تر می شود چرا که نمی تواند خطای های زیر حد نرخ یادگیری را تغییر دهد.

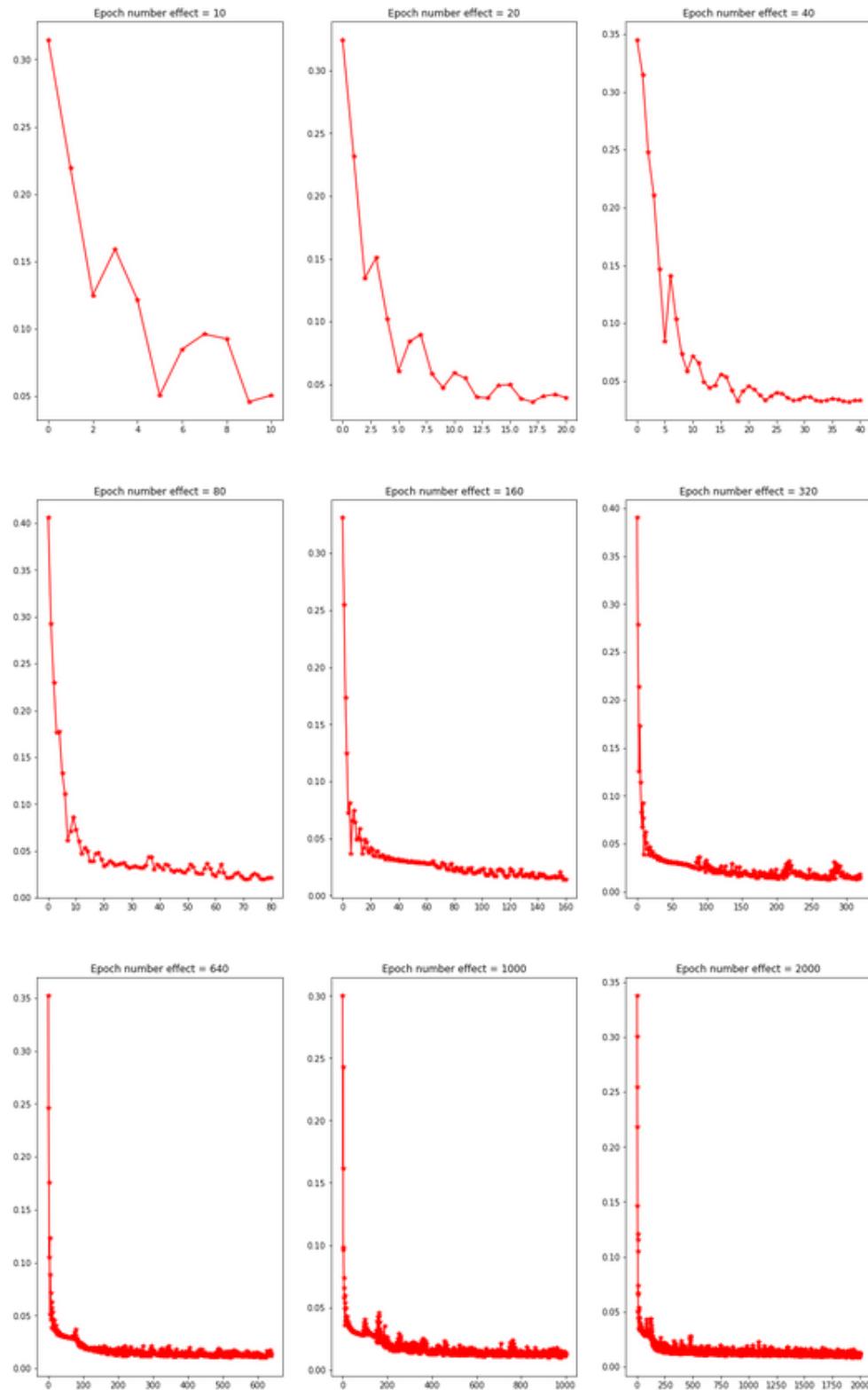
البته گاهی وقتا اگر بیش از حد نرخ یادگیری را تغییر بدھیم و آگرا می شود!!! که همانطور که مشاهده می کنید دو سطر آخر نمودارها هویداگر چنین رخ دادی اند.

• قسمت اثر :batch Size

همانگونه در ادامه می بینیم در فرآیند یادگیری هر بار بسته ی داده ها به شبکه داده می شود و هر چقدر این بسته ی داده ها بزرگ باشند داریم که سیر خطای زودتر همگرا به صفر می گردد. (چرا که وقتی سایز بسته ها بیشتر باشد میزان داده های در کنار هم برای ترین شبکه افزایش یافته و شبکه بهتر یاد می گیرد و خطای حاصل کاهش می یابد...)

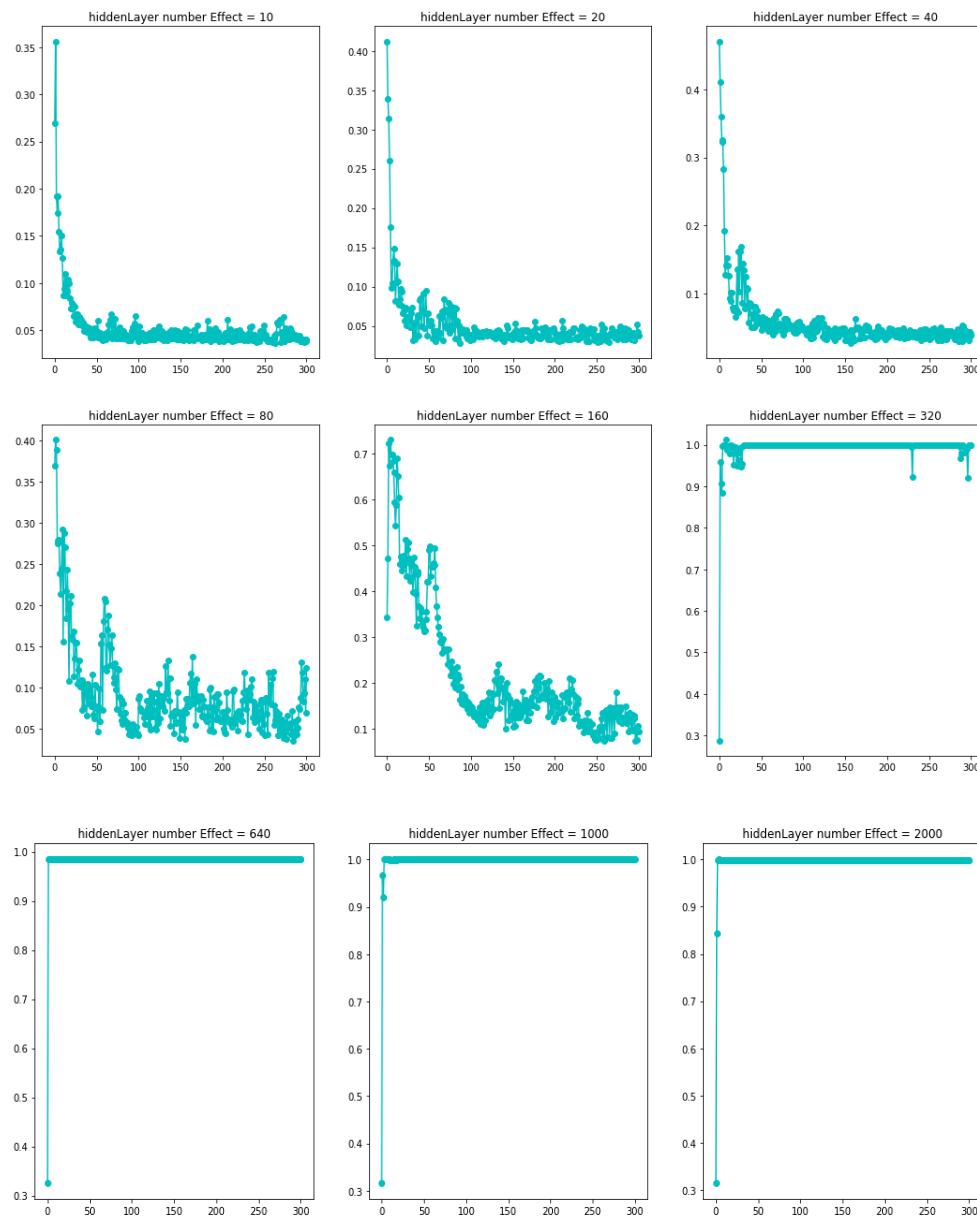


قسمت اثر : number of epochs •



نتیجه گیری: همانطور که مشاهده می کنید همانند نمودار های سوال های قبل در ارتباط با number of epochs داریم که در اثر بیشتر شدن تعداد تکرار، داریم که تاثیرگذارتر از تغییر سایر پارامتر های شبکه، شبکه همگرا می شود و مثلا برخلاف learning Rate داریم که در اثر افزایش زیاد تکرار واگرایی رخ نمی دهد لذا برای یک شبکه عصبی داریم که از بهترین کارها افزایش مراتب تکرار می باشد تا شبکه بهتر یاد بگیرد و برخلاف سایر پارامتر های شبکه که ممکن است تغییرشان اثر فاحشی رو واگرای یا هم گرایی بگذارند داریم که تغییر number of epochs صرفا تعداد تکرار را زیاد می کند و کمک می کند شبکه هر چه بهتر یاد گرفته شود.

• قسمت اثر hidden Layer Effect •



نتیجه‌گیری:

همانطور که میدانیم پیچیدگی تابع یادگیری شده است قسمت نسبت به قسمت های قبل بیشتر می باشد لذا افزایش تعداد نورون ها تا رسیدن به یک حد معقولی باعث می شوند که فضای حالت شبکه بیشتر می شود و تابع بهتر و دقیق تر یادگرفته می شود.

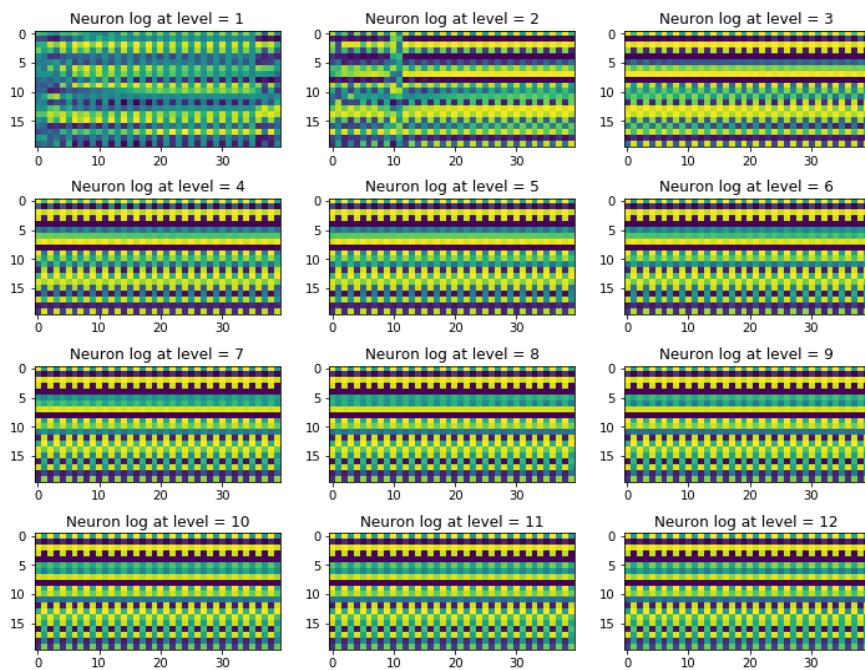
البته توجه کنید که همانطور که گفتم افزایش تا حد معقول! و اگر از حد زیادی تعداد نورون های میانی بیشتر شود داریم که مساله overfitting ممکن است رخ بدهد و صد البه نشت خطای ناچیز سبب واگرایی شدید سامانه شود لذا با توجه به سیر نمودارها داریم که تعداد های نورون ها (هر چه به 10 نزدیک تر بهتر) برای یادگیری شبکه مناسب است اما افزایش تعداد از این حد باعث پراکندگی خطا می گردد و اگر از یه حدی بیشتر تعداد نورون ها زیاد شود شبکه یاد نمی گیرد و واگرایی رخ می دهد.

بخش 9: فعالیت نورون ها

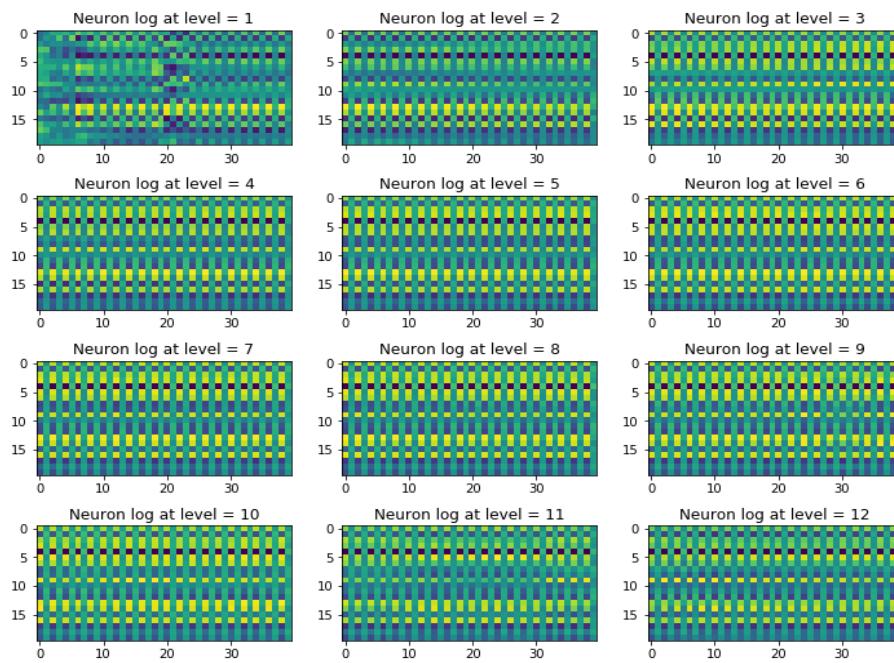
توجه کنید که نمودار های صفحه‌ی بعد نشان می دهید که چقدر فعالیت نورون ها به مقادیر وزن ها در ابتدا بستگی دارد و داریم که در اصل می دانیم که تعداد نورون ها زیاد می باشد و داریم که عملکرد و فعالیت شبکه در نهایت منتها به همین فعالیت های نورون ها دارد و لذا داریم که تغییرات در وزن ها باعث تغییرات در فعالیت نورون ها می گردد و افزایش یافتن یک وزن منتها به فعال تر شدن فعالیت نورون مرتبط با آن وزن می گردد حال آنکه کمتر شدن یک وزن حین آموزش منجر به کاهش فعالیت نورون مربوط به آن وزن می گردد و داریم که این تغییر فعالیت ها به کمک رنگ ها و transient رنگ ها در نمودار های صفحه‌ی بعد مشخص تر است.

$$\text{Learning Rates} = [0.001, 0.005, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.5, 1]$$

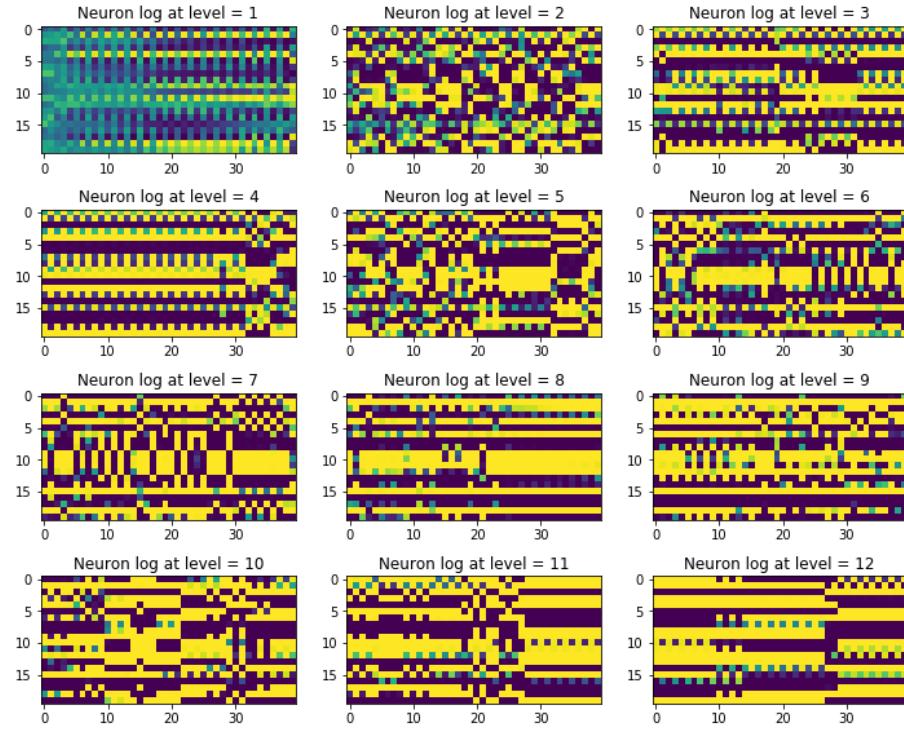
learning Rate is 0.001000



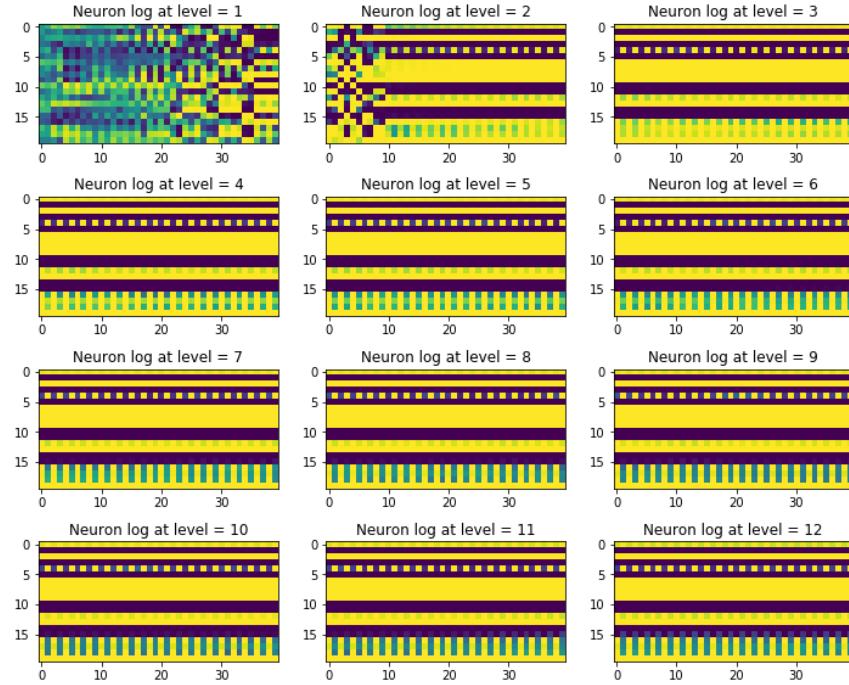
learning Rate is 0.005000



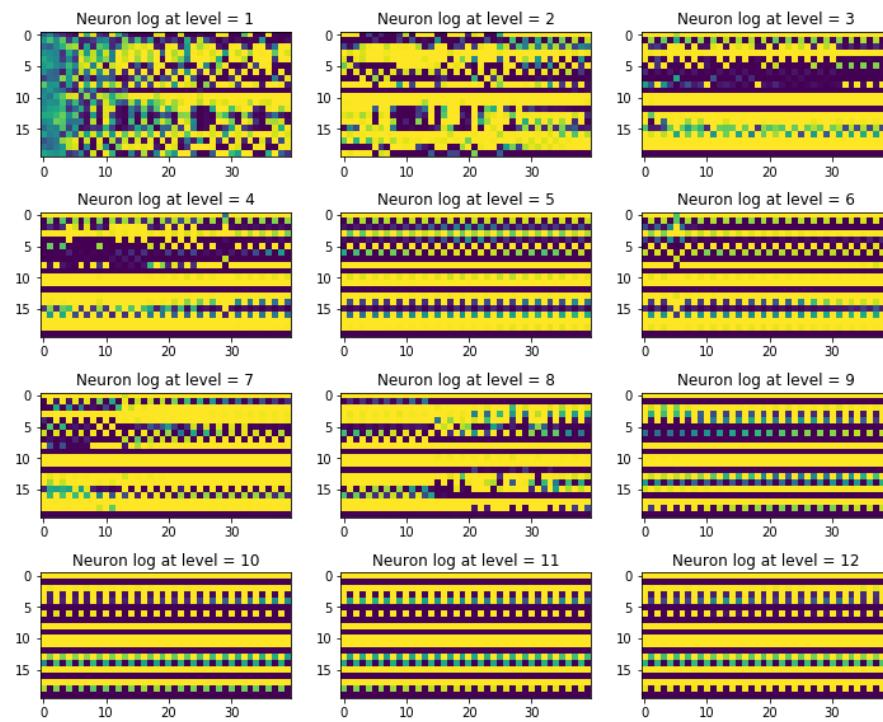
learning Rate is 0.010000



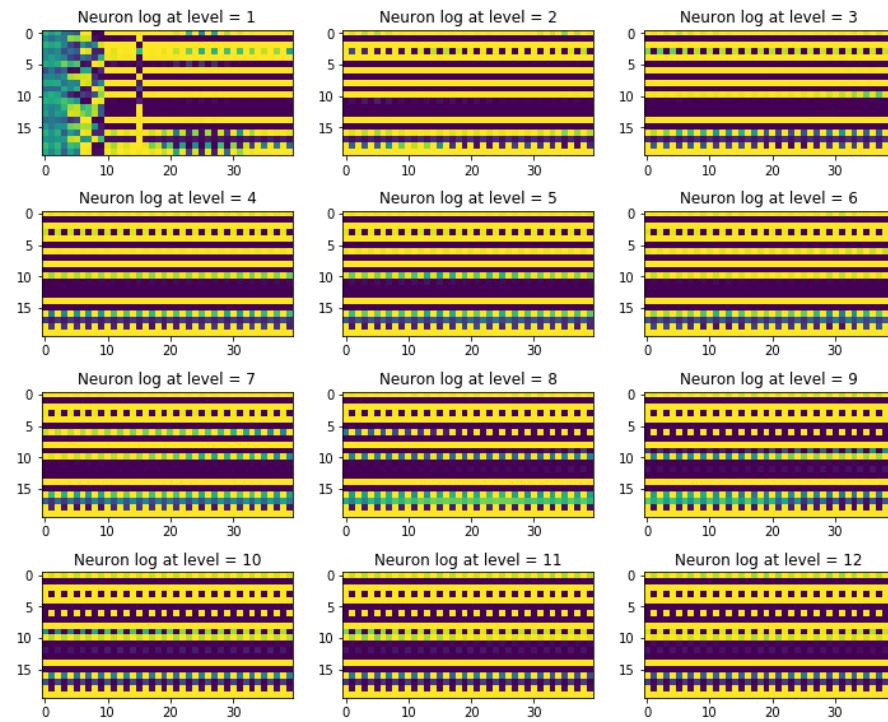
learning Rate is 0.020000



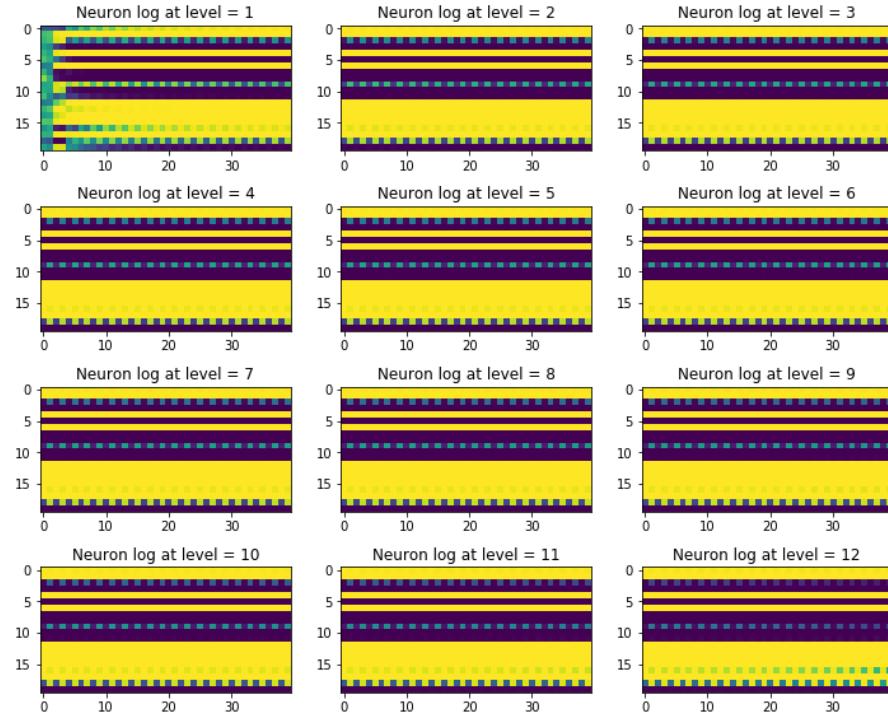
learning Rate is 0.040000



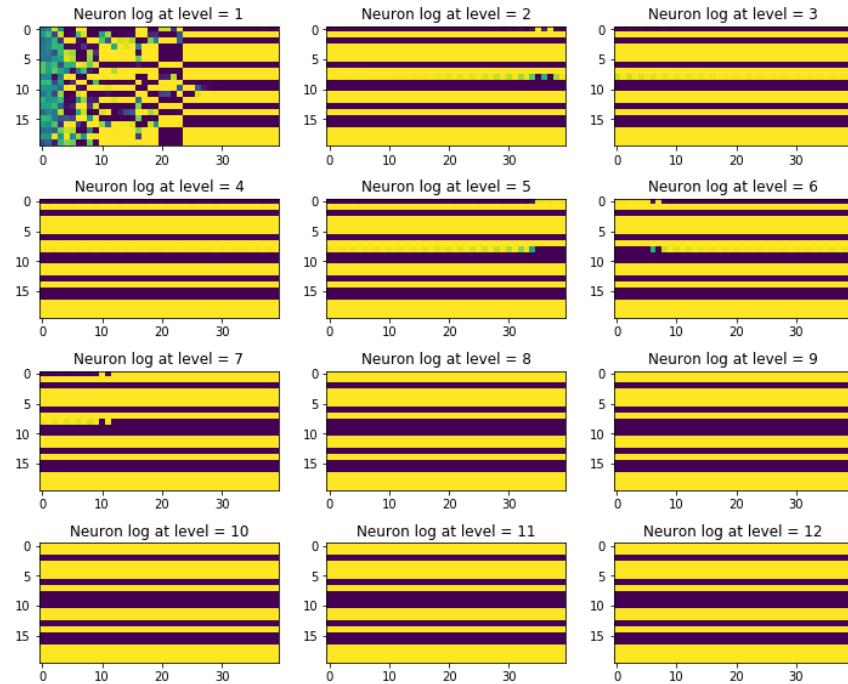
learning Rate is 0.080000



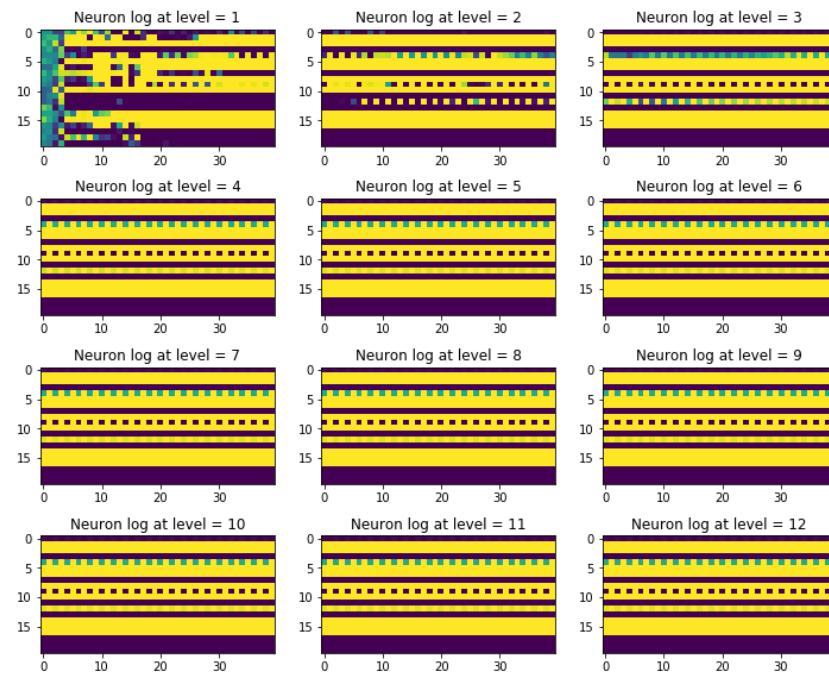
learning Rate is 0.160000



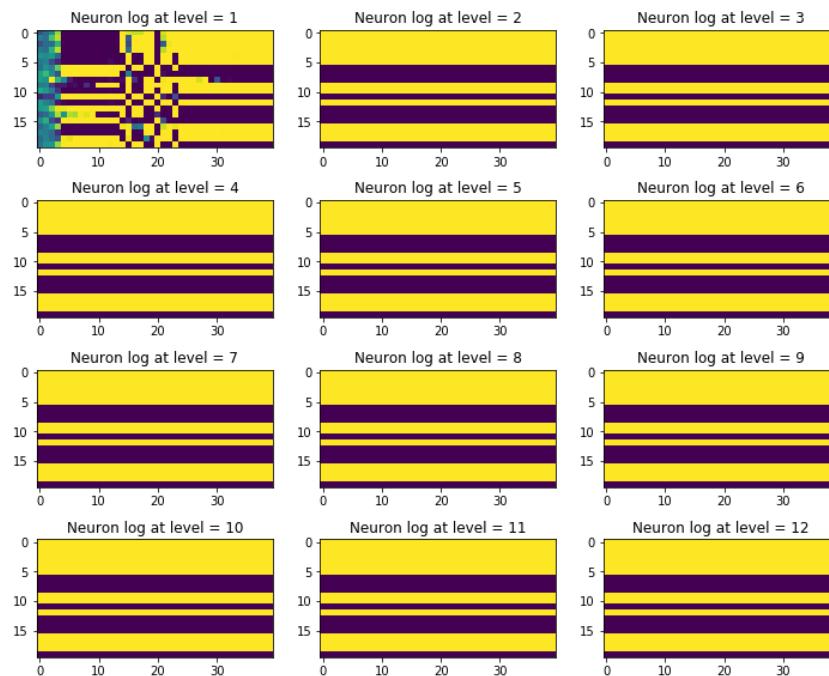
learning Rate is 0.320000



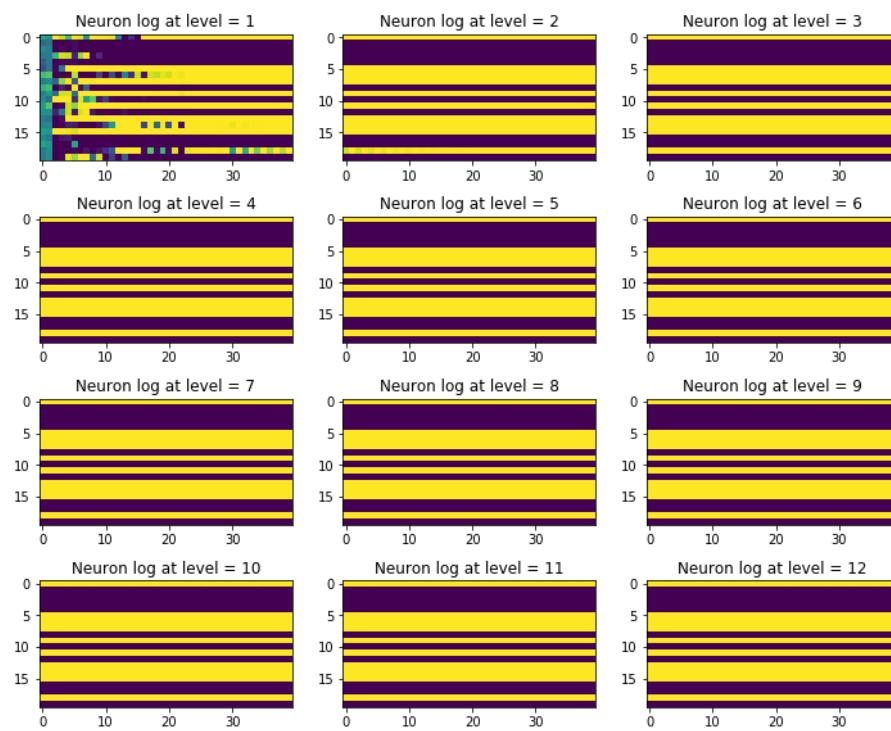
learning Rate is 0.500000



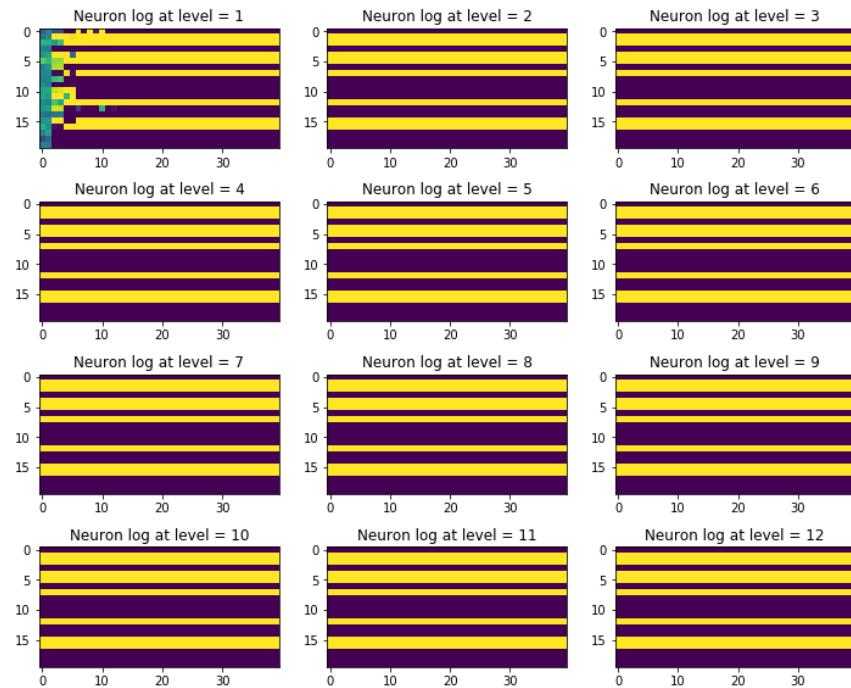
learning Rate is 1.000000



learning Rate is 1.500000



learning Rate is 2.000000



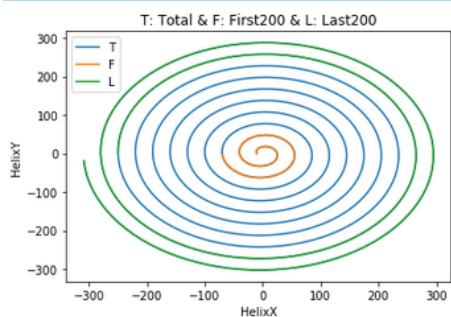
سوال چهارم:

بخش اول:

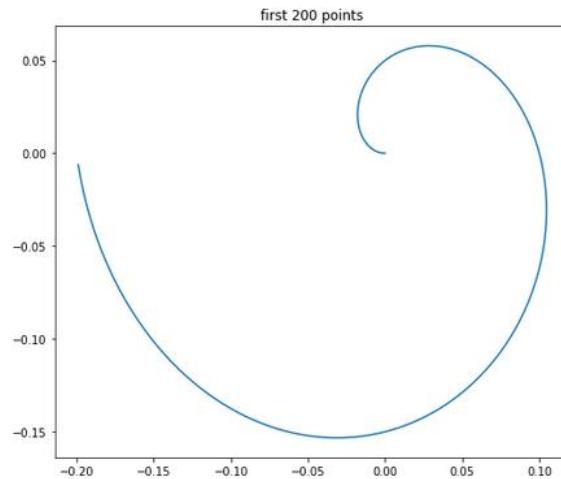
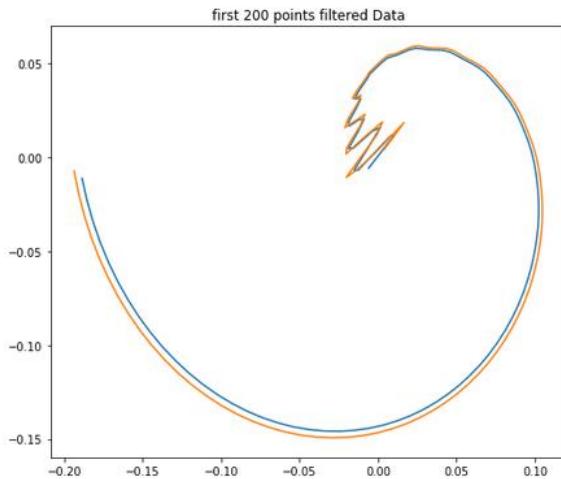
ابتدا طبق گفته‌ی سوال خم دو بعدی مد نظر را می‌کشیم:

توجه کنید که رنگ سبز 200 مورد آخر و رنگ زرد 200 مورد اول می‌باشد و رنگ آبی 200 تا 800 امین داده می‌باشد.

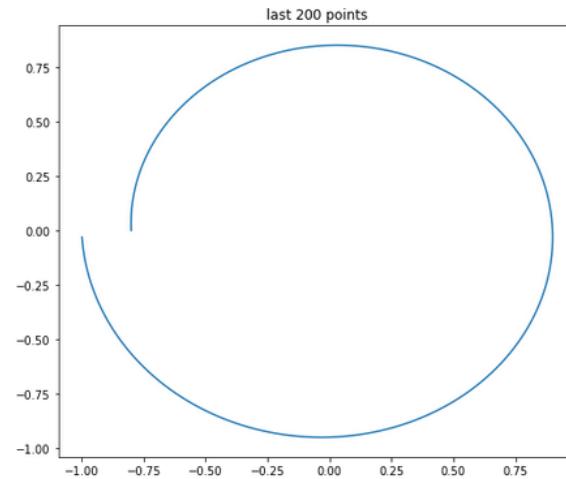
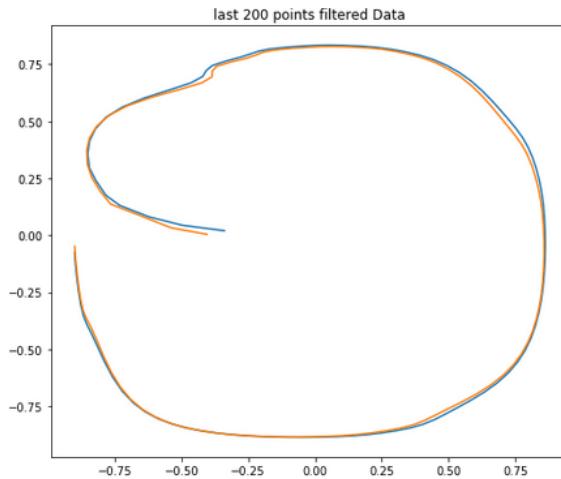
```
1 #
2 # first last see what is the helix!!!! and its first and last 200 indexes ...
3 # in the code below we'll show first 200 and last 200 elements of this helix and also its Complete shape.
4 #
5 import numpy as np
6 import matplotlib.pyplot as plt
7 t = np.arange(1000)
8 u = 0.3
9 r0 = 10
10 r = r0 + u * t
11 omega = 2 * np.pi * 0.01
12 phi0 = np.pi
13 phi = -omega * t + phi0
14 x = r * np.cos(phi)
15 y = r * np.sin(phi)
16
17 # total Helix ...
18
19 plt.plot(x,y)
20
21 # first 200 elements ...
22 x_first200 = x[:200]
23 y_first200 = y[:200]
24 plt.plot(x_first200, y_first200)
25
26 # last 200 elements ...
27 x_last200 = x[800:]
28 y_last200 = y[800:]
29 plt.plot(x_last200,y_last200)
30 plt.legend('TFL')
31 plt.title("T: Total & F: First200 & L: Last200")
32 plt.ylabel("HelixY")
33 plt.xlabel("HelixX")
34 plt.show()
35 plt.gcf().set_size_inches(18, 60)
36
37
```



تست کردن 200 داده ای اول:

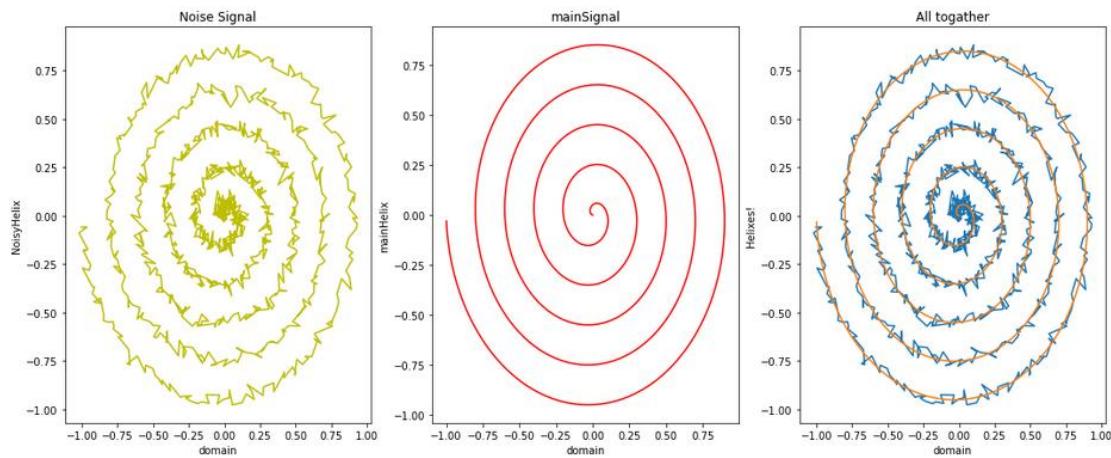


تست کردن 200 داده ای انتهایی:



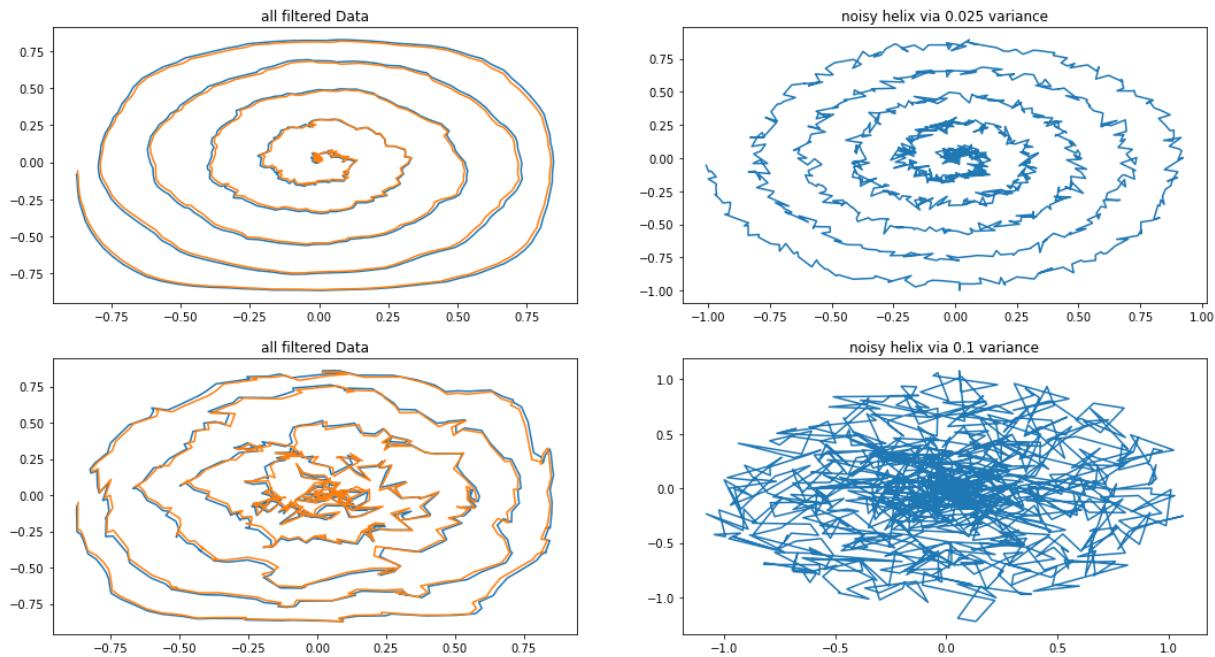
بخش دوم:

داده ای نویزی با واریانس 0.1 و میانگین صفر به مارپیچ خود اضافه می کنیم.



بخش سوم و چهارم همگی باهم:

در ستون سمت راست داریم که مارپیچ با نویز بوده و سمت چپ داریم که خروجی سیستم می باشد.



: SNR مقدار

در حالت اول با واریانس 0.025 داریم که SNR برابر است با مقدار زیر:

$$\text{SNR} = 0.6464400581814659$$

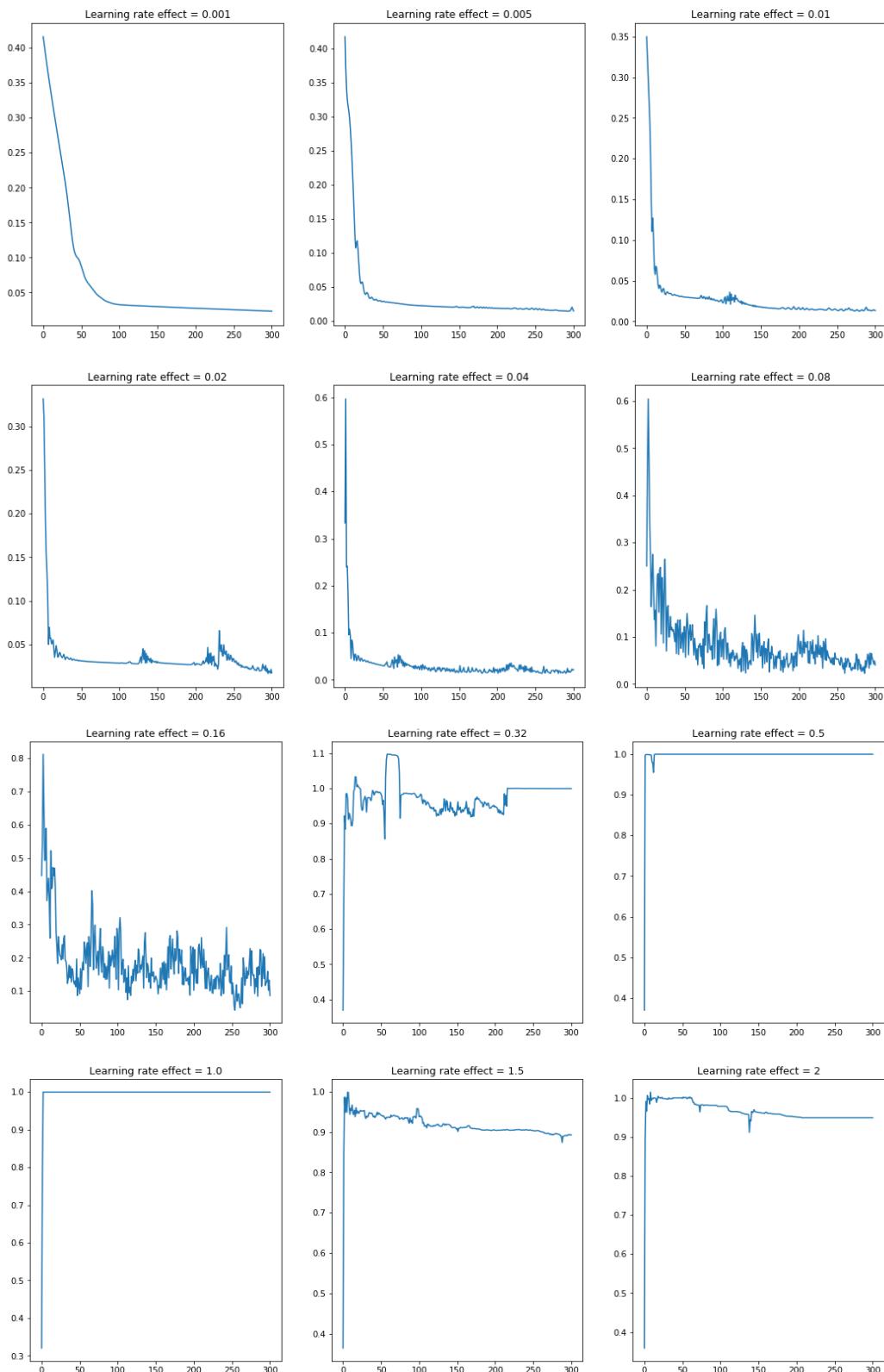
و در حالت دوم با واریانس 0.1 داریم که SNR برابر است با مقدار زیر:

$$\text{SNR} = 0.6523117969279401$$

با افزایش واریانس داریم که میزان SNR نیز افزایش پیدا می کند و این خود باعث می شود که سیگنال فیلتر شده به سیگنال اصلی نزدیک تر شود (می توانید به نمودارهای Sin سوال قبل و سیر افزایش واریانس آنها توجه کنید) حال آنکه داریم اگر واریانس داده های خیلی زیاد شود خواهیم داشت که در نهایت که از یه حدی داده ها بیشتر خراب می شوند دیگر شبکه قادر به بازسازی نسبتا خوب مارپیچ نمی باشد و این سیر به نوعی یک threshold را نشان می دهد که از آن پس داریم که میزان بازسازی داده های نویزی شبکه نسبتا ضعیف می باشد...

بخش پنجم:

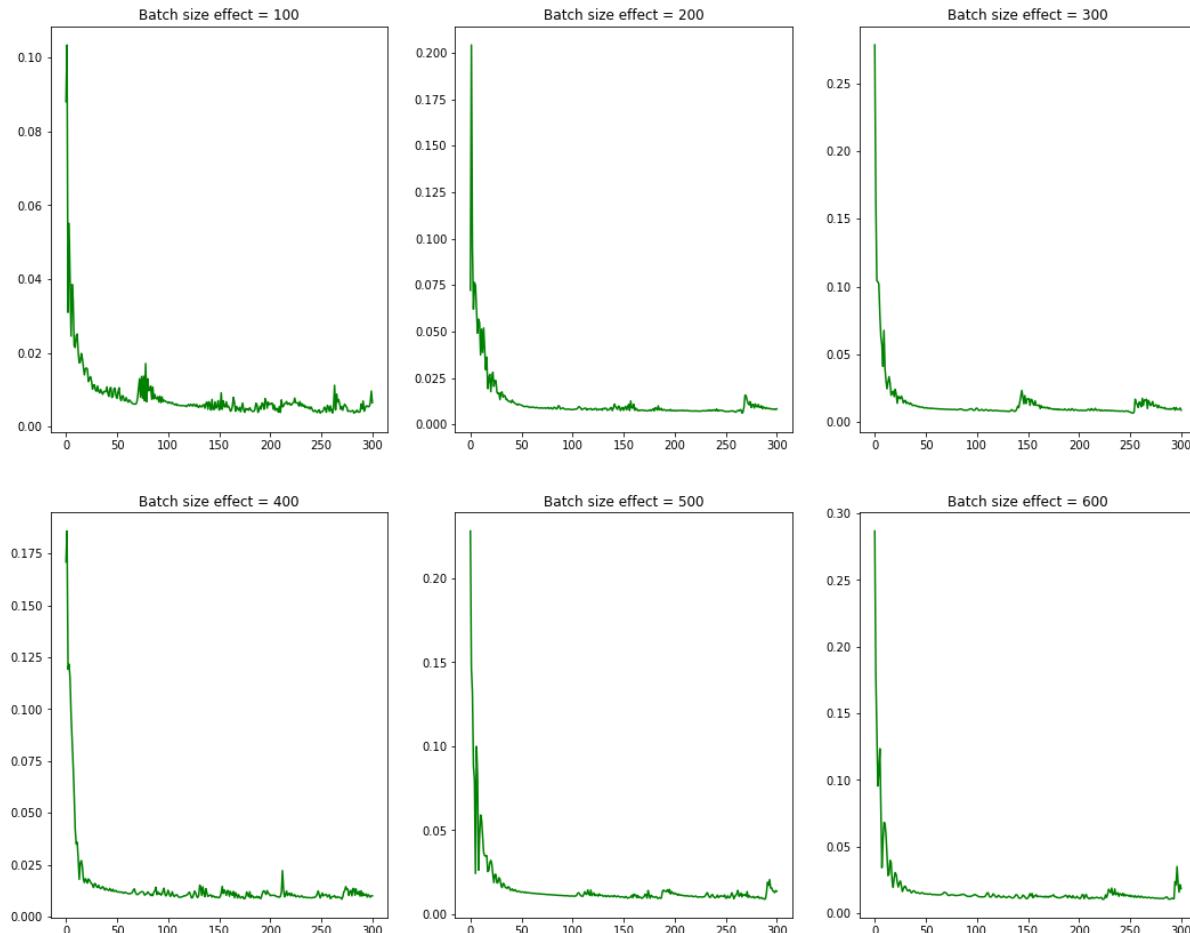
قسمت اول: learning Rate Effect •



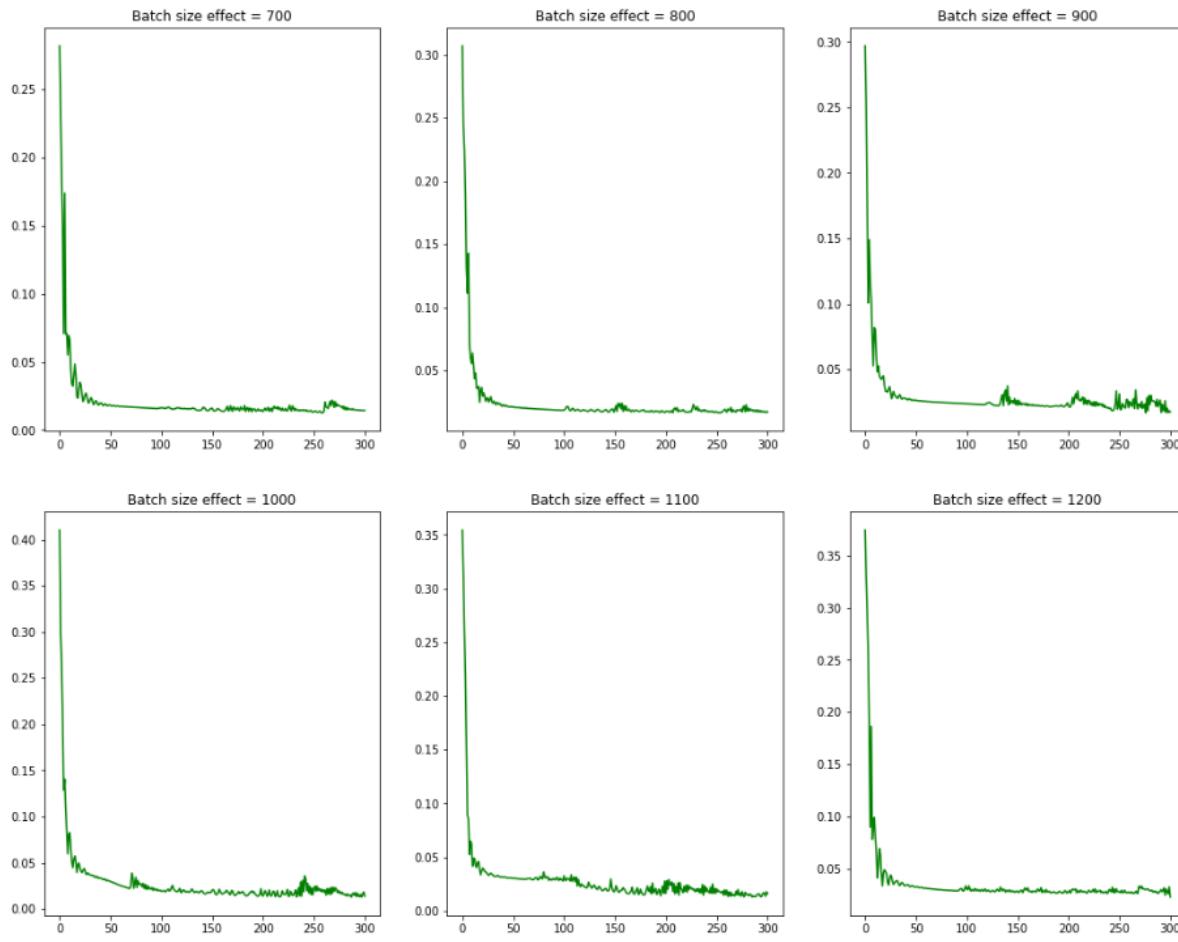
نتیجه گیری: همانطور که مشاهده می کنید داریم

- برای مقادیر کمتر نرخ یادگیری: یادگیری دیر و نرخ loss همگرا ○
- برای مقادیر متوسط نرخ یادگیری: یادگیری زود و نرخ loss همگرا ○
- برای مقادیر زیاد نرخ یادگیری: واگرایی ○

قسمت دوم: بررسی اثر batch size •



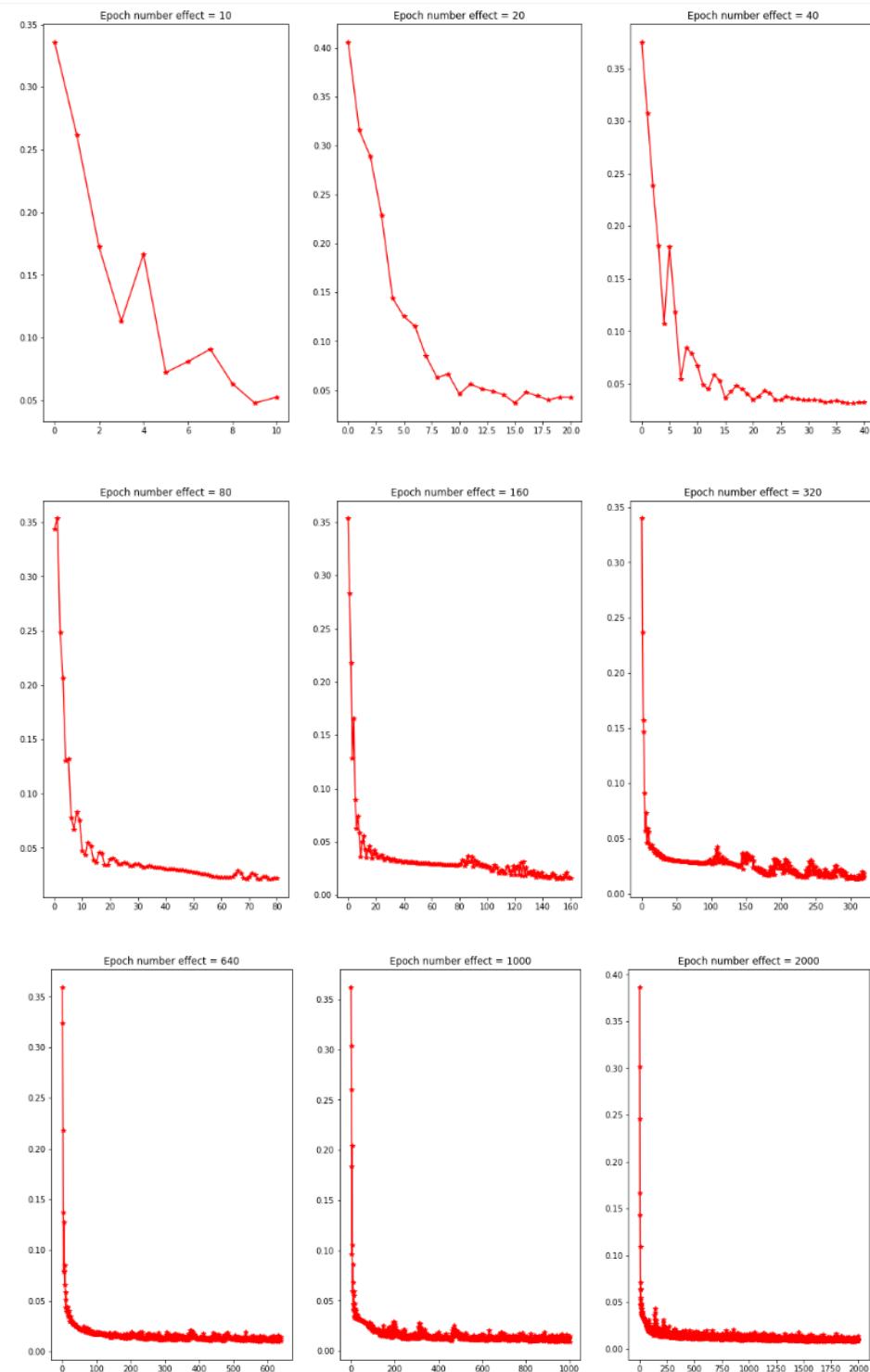
ادامه‌ی سیر batch size effect را در صفحه‌ی بعد مشاهده می‌کنیم:



نتیجه گیری:

در کمال تعجب داریم که با حتی چندین برابر شدن batch size ها از حدود 100 تا 1200 (12 برابر شدن) با هم هم چنان شبکه با قوت تمام می تواند فیلترینگ را انجام دهد هر چند حجم داده های تحویلی به شبکه ی عصبی در طی این مثال ها خیلی تغییر میکند اما هم چنان با قدرت شبکه عمل فیلترینگ را انجام می دهد که این قدرت بالا را مرهون بازگشتنی بودن شبکه می باشد حال آنکه تغییر سایز batch در شبکه های غیر بازگشتنی به مراتب اثر مشخص تری در واگرایی هم گرایی دارد.

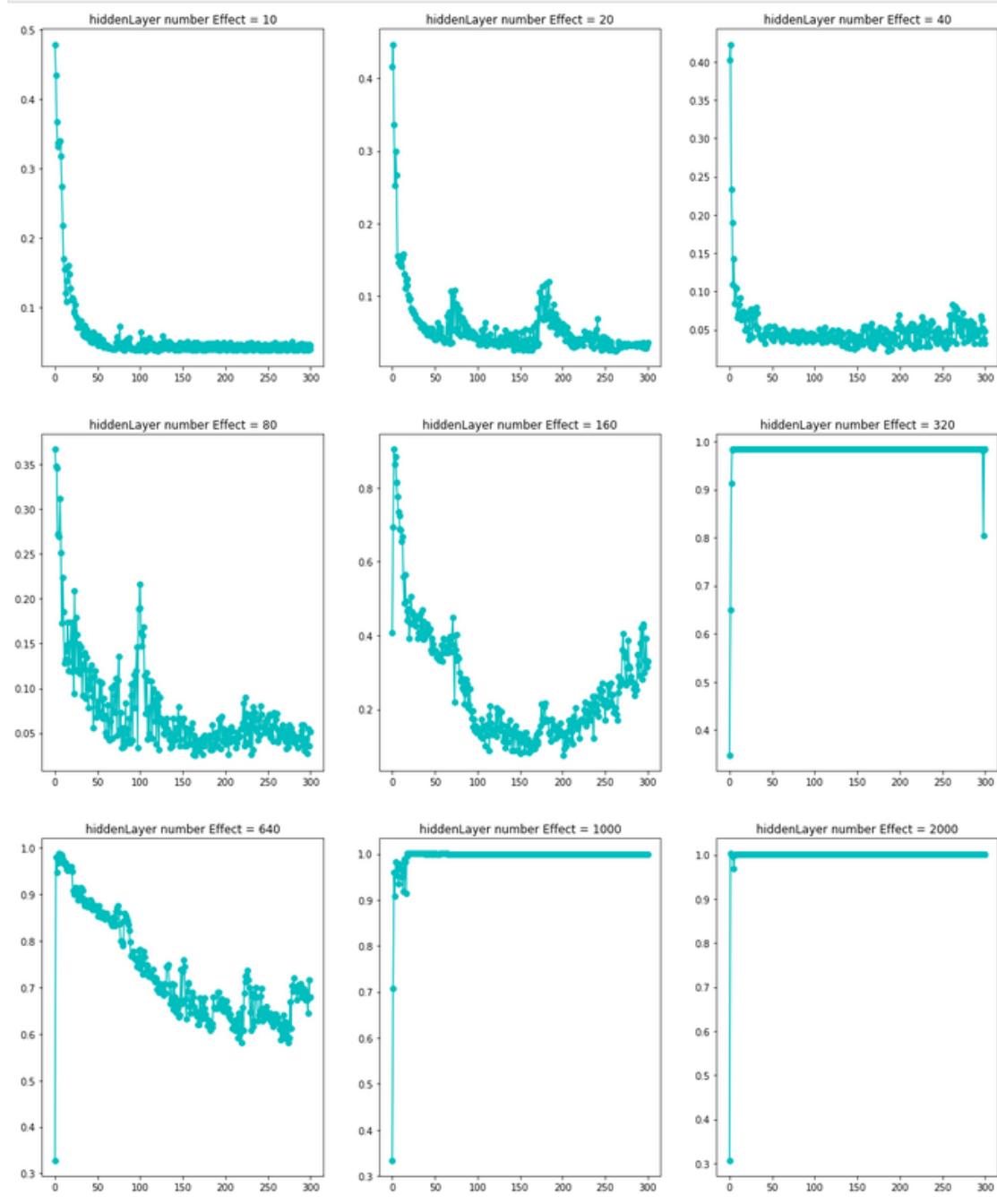
قسمت سوم: بررسی اثر loss of epochs •



جزیيات نتیجه گیری در صفحه‌ی بعد آورده شده است.

نتیجه گیری: همانطور که مشاهده می کنید همانند نمودار های سوال های قبل در ارتباط با number of epochs داریم که در اثر بیشتر شدن تعداد تکرار، داریم که تاثیرگذارتر از تغییر سایر پارامتر های شبکه، شبکه همگرا می شود و مثلا بر خلاف learning Rate داریم که در اثر افزایش زیاد تکرار واگرایی رخ نمی دهد لذا برای یک شبکه‌ی عصبی داریم که از بهترین کارها افزایش مراتب تکرار می باشد تا شبکه بهتر یاد بگیرد و بر خلاف سایر پارامتر های شبکه که ممکن است تغییرشان اثر فاحشی رو واگرای یا هم‌گرایی بگذارند داریم که تغییر number of epochs صرفا تعداد تکرار را زیاد می کند و کمک می کند شبکه هر چه بهتر یاد گرفته شود.

قسمت چهارم: بررسی اثر hiddenLayerSize



نتیجه گیری:

همانطور که میدانیم پیچیدگی تابع یادگیری شده است قسمت نسبت به قسمت های قبل بیشتر می باشد لذا افزایش تعداد نورون ها تا رسیدن به یک حد معقولی باعث می شوند که فضای حالت شبکه بیشتر می شود و تابع بهتر و دقیق تر یادگرفته می شود.

البته توجه کنید که همانطور که گفتم افزایش تا حد معقول! و اگر از حد زیادی تعداد نورون های میانی بیشتر شود داریم که مساله overfitting ممکن است رخ بدهد و صد البته نشت خطای ناچیز سبب واگرایی شدید سامانه شود لذا با توجه به سیر نمودارها

داریم که تعداد های نورون 10 الی 40 (هر چه به 15 نزدیک تر بهتر) برای یادگیری شبکه مناسب است اما افزایش تعداد از این حد باعث پراکندگی خطای گردد و اگر از یه حدی بیشتر تعداد نورون ها زیاد شود شبکه یاد نمی گیرد و واگرایی رخ می دهد.

بخش ششم: اثر فعالیت نورون ها

ابتدا ذکر مطلب زیر مهم است: (برداشته شده از سورس کد این قسمت)

```
# since I calculate different RNN in the part 5.2
```

```
# in this part I Used the data calculated in first part of 5.2
```

```
# also known as D1 learning Rate effect ...
```

```
# but in the first part of 5.2 we sweep among this learning rates ...
```

```
# learning rate = [0.001, 0.005, 0.010, 0.020, 0.040, 0.080, 0.160, 0.32, 0.50, 1.0, 1.5, 2]
```

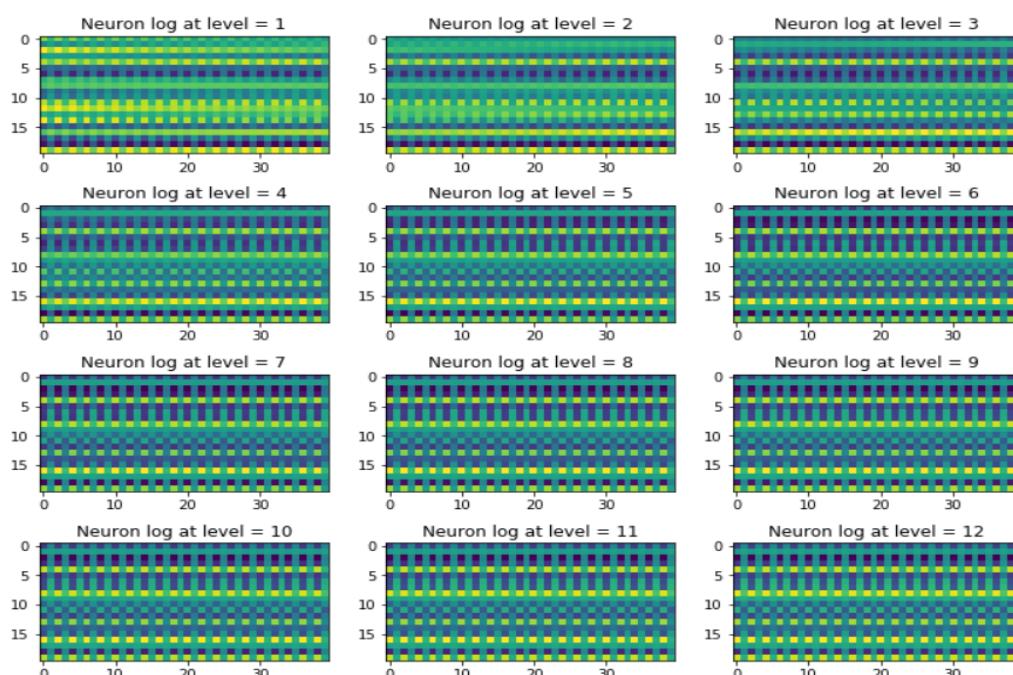
```
# so I created a for via I in range of 1 until 12 and use D1[i]
```

```
# with the learning rate of learningRate[i]
```

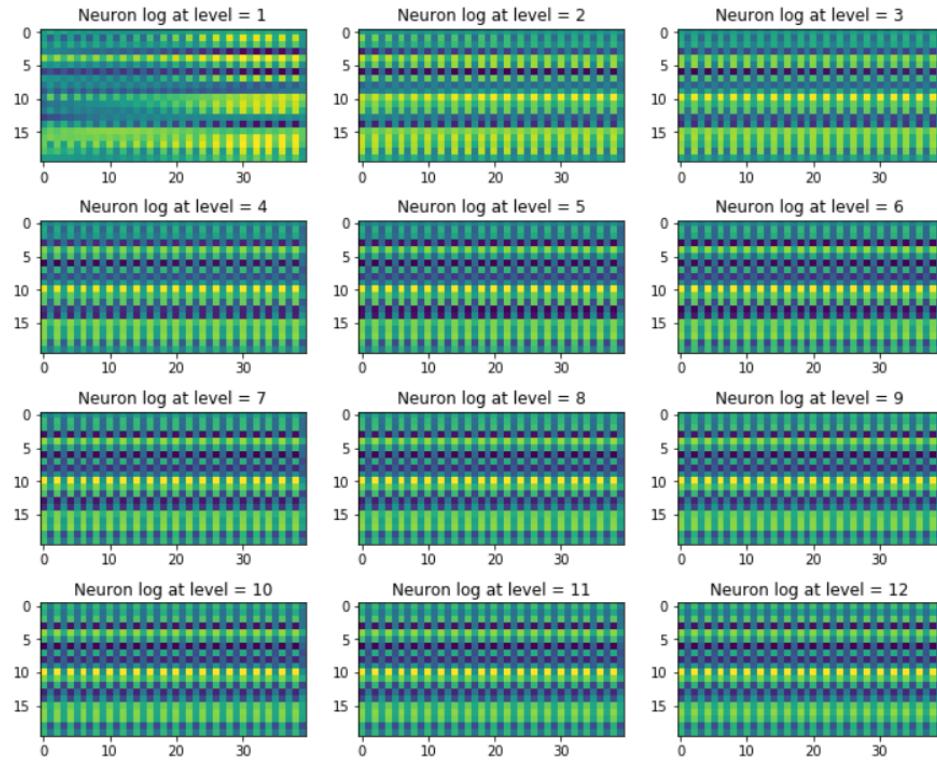
در صفحه‌ی بعد به نمودار های این بخش می پردازیم:

Learning Rate = 0.001

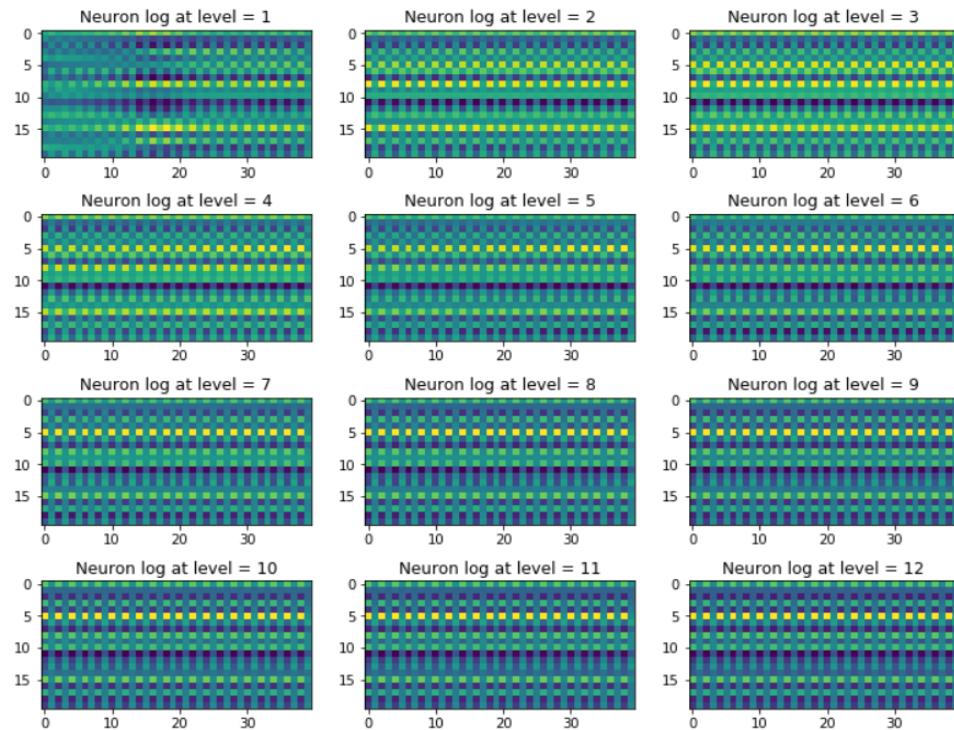
```
learning Rate is 0.001000
```



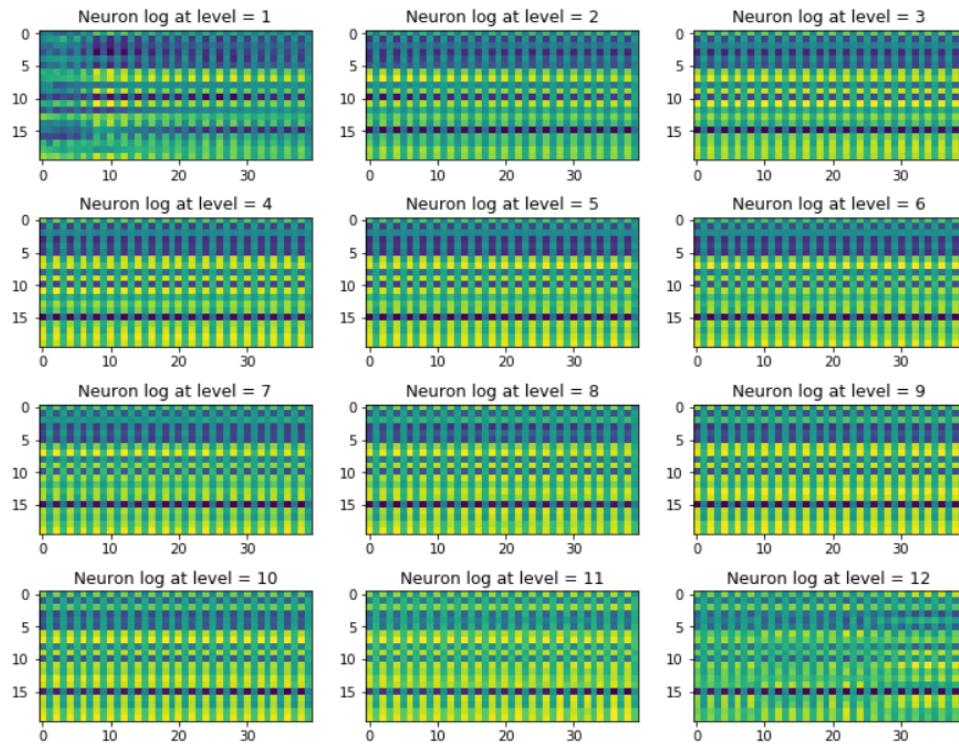
learning Rate is 0.005000



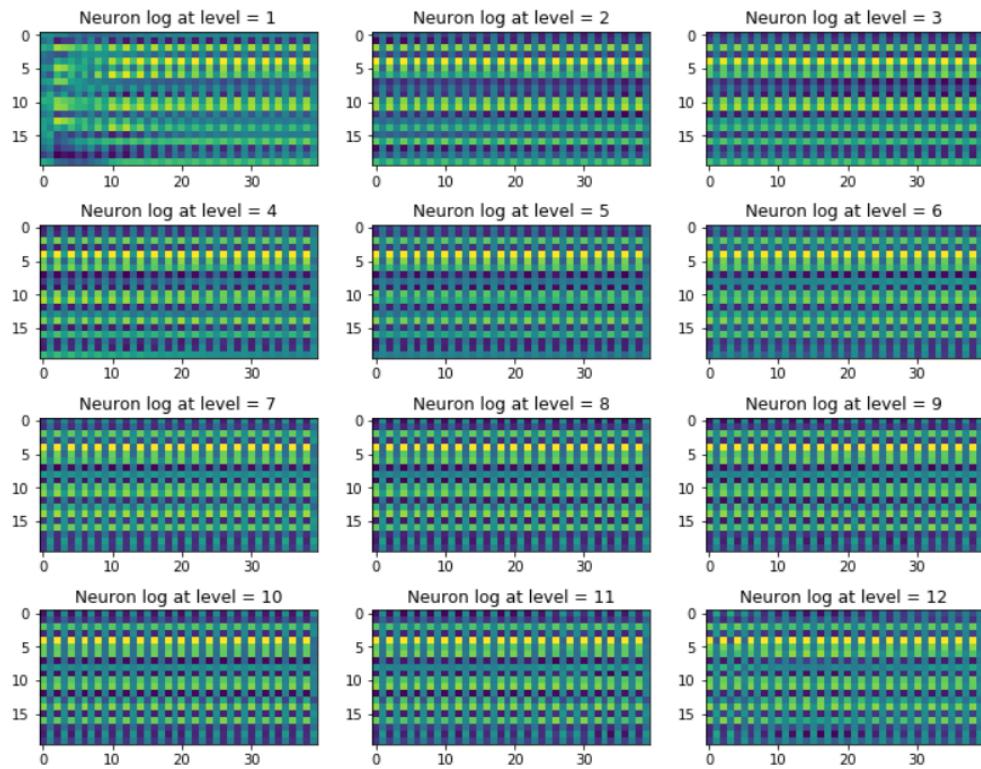
learning Rate is 0.010000



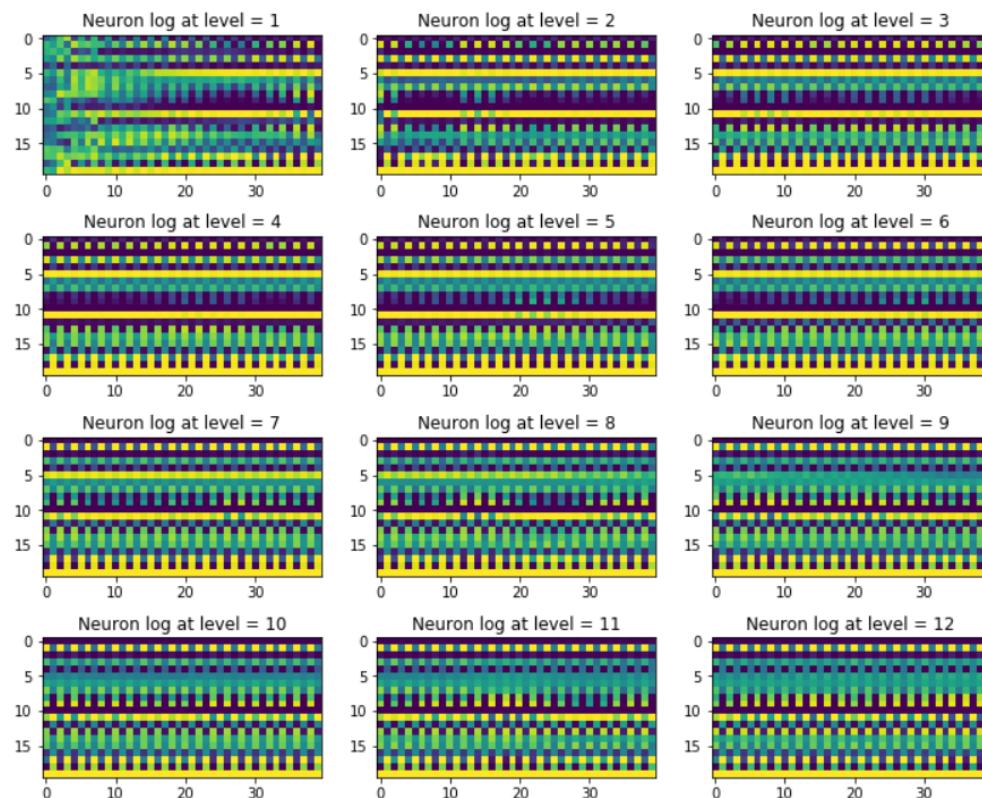
learning Rate is 0.020000



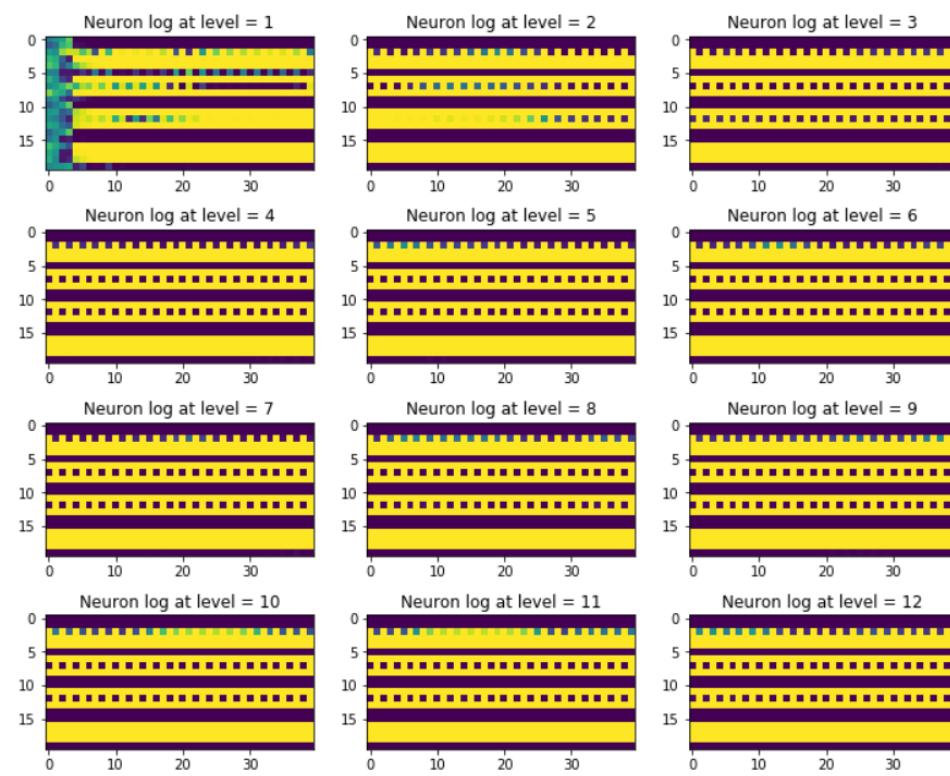
learning Rate is 0.040000



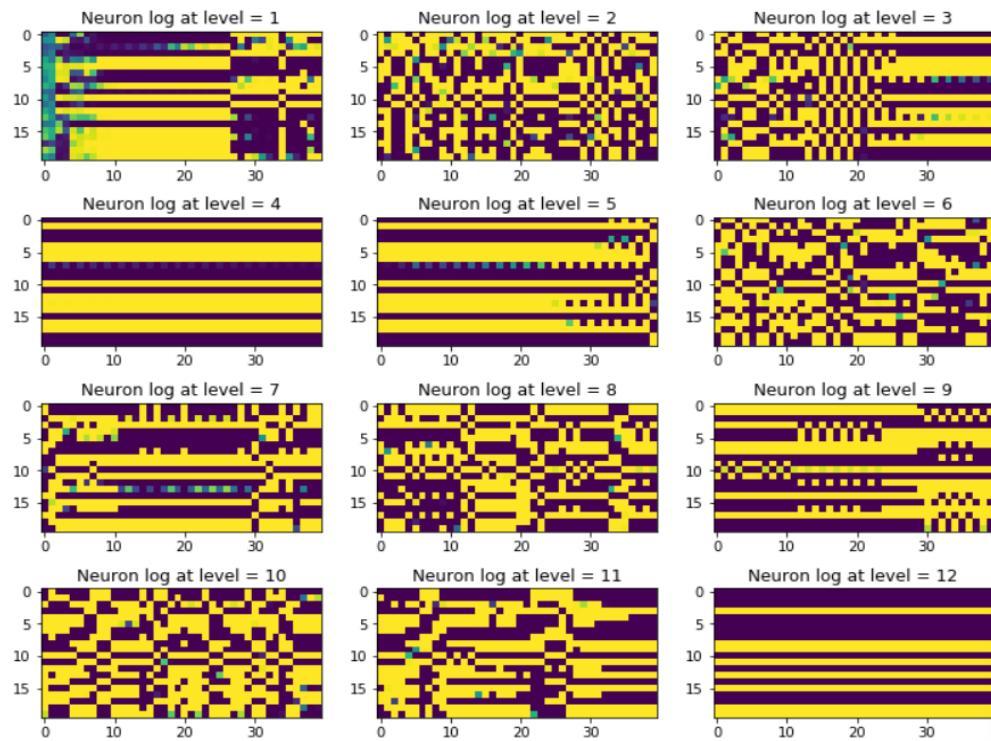
learning Rate is 0.080000



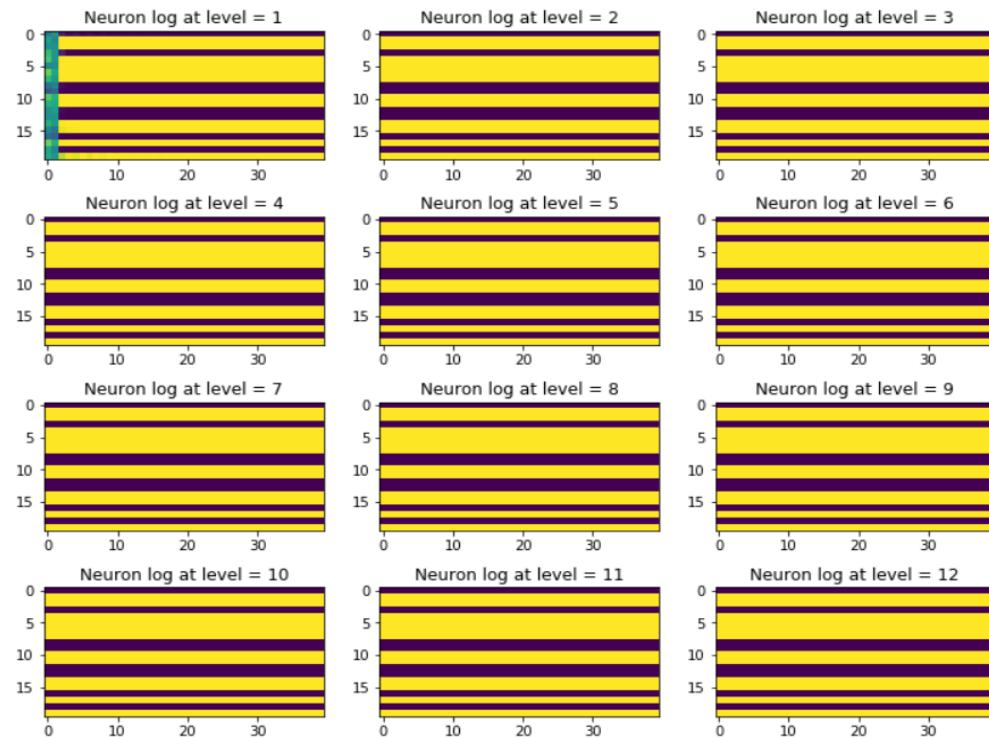
learning Rate is 0.160000



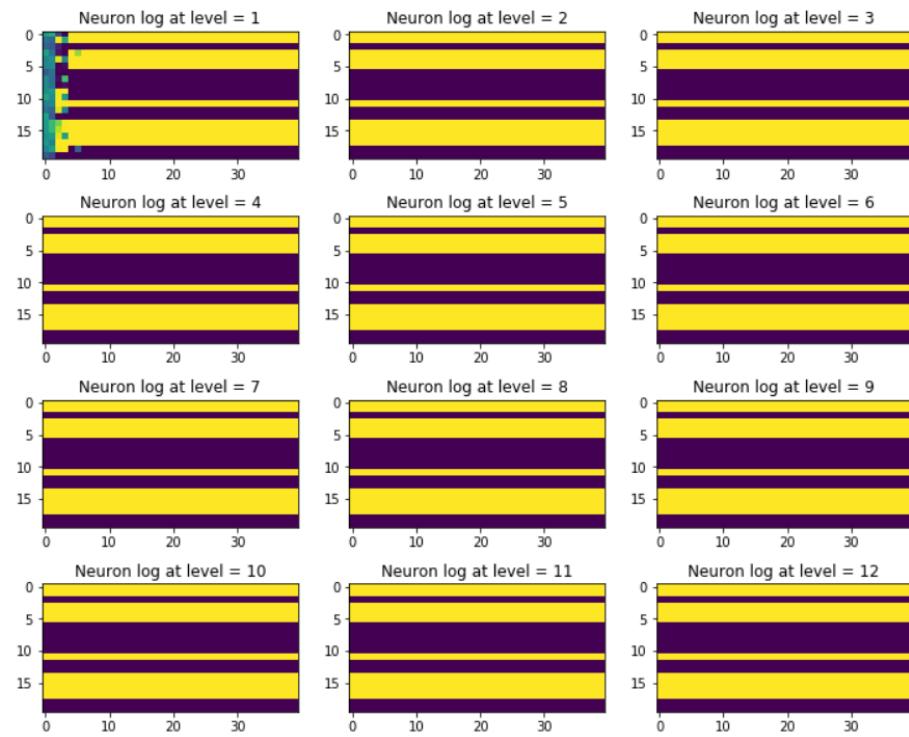
learning Rate is 0.320000



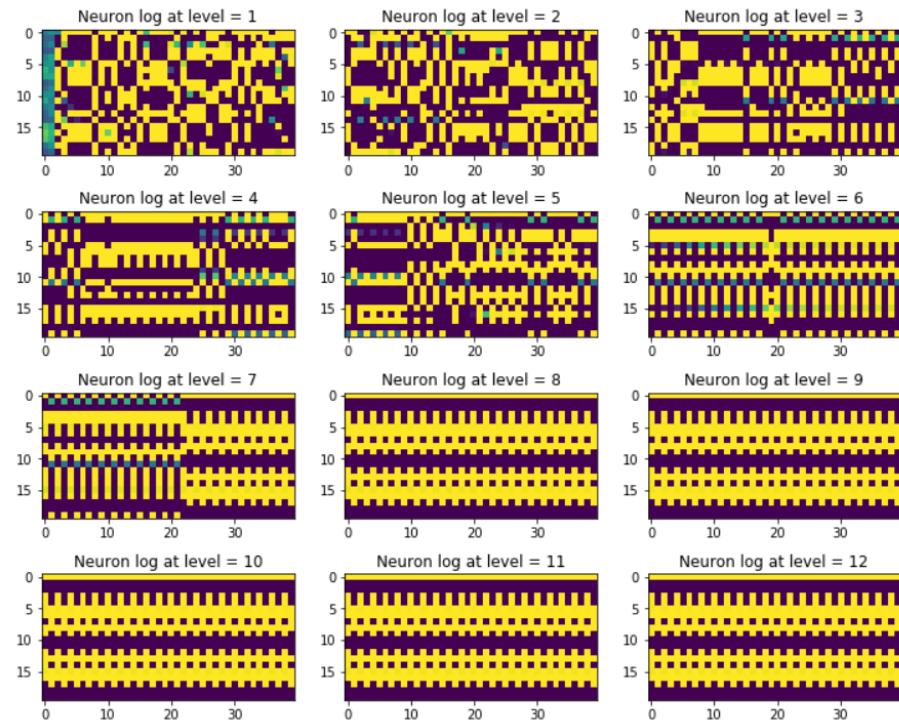
learning Rate is 0.500000



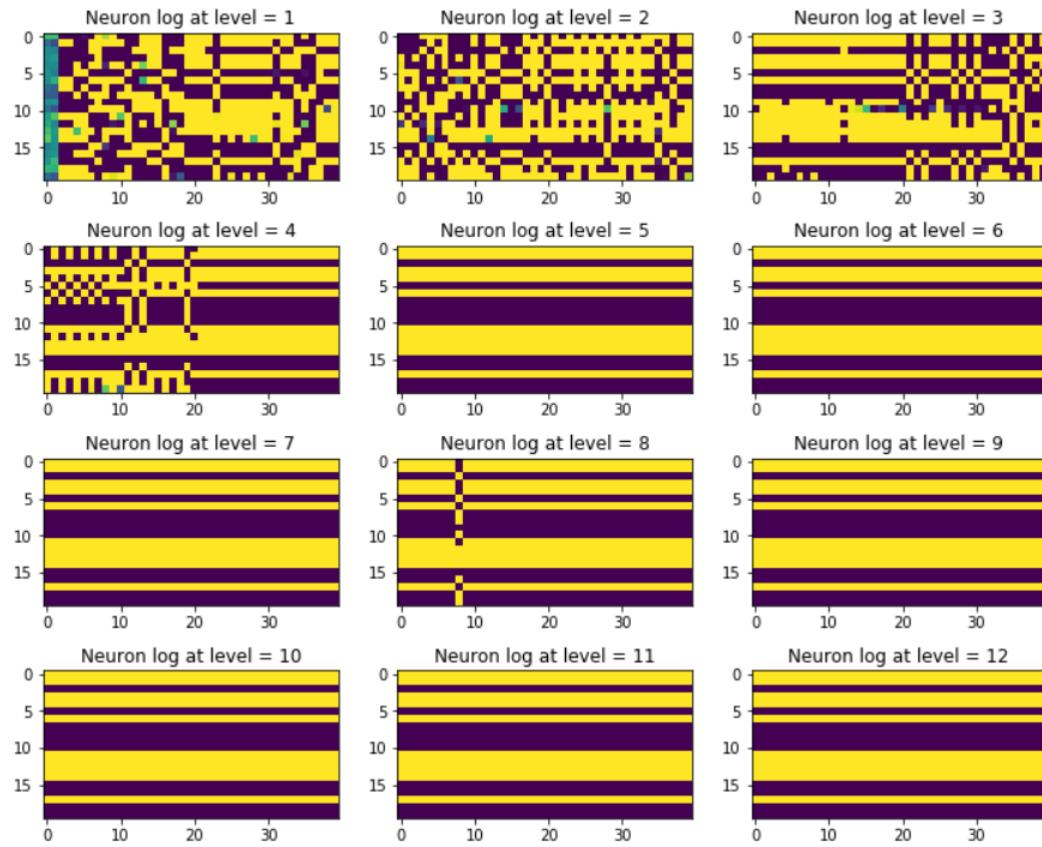
learning Rate is 1.000000



learning Rate is 1.500000



learning Rate is 2.000000



نتیجه گیری :

همانند بخش های قبل داریم که نورون ها به شدت به شرایط و حالت اولیه شان وابستگی رفتاری دارند و پس از شروع پس از گذر زمان بالاخره و گاه ناگهانی به حالت پایدار می رسند(در واقع حساسیت خود تنها معطوف داده های جدید می کنند و تقریباً دیگر تحت تاثیر داده های قبلی نمی باشند) و پس از آن تغییر رفتاری در آنها مشاهده نمی گردد و دیدن این رفتار در سلسله تصاویر بالا به وضوح مشخص می باشد.