به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

تمرین سری 3

نام و نام خانوادگی	شایان واصف احمدزاده
شماره دانشجویی	810197603
تاریخ ارسال گزارش	1 دی 1400

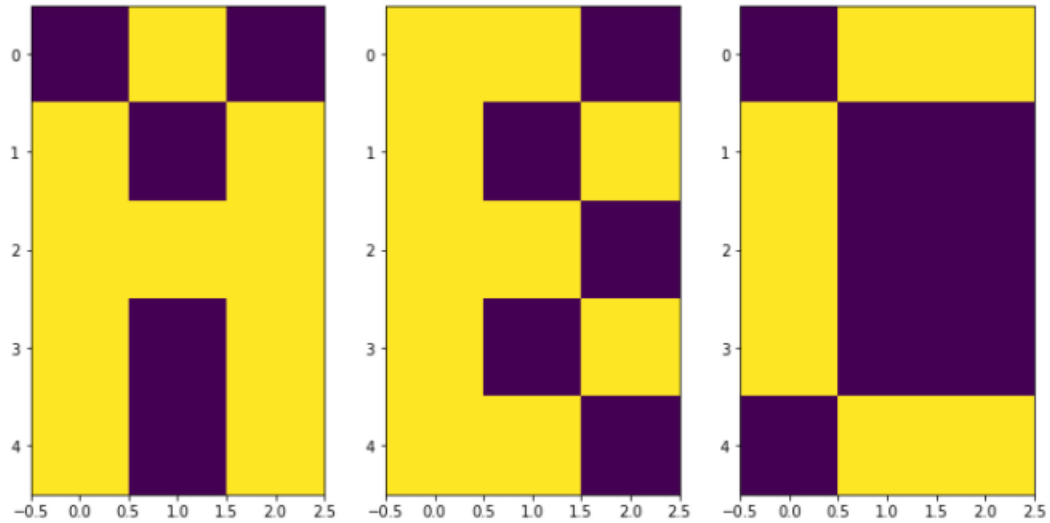
فهرست گزارش سوالات

سوال 1 – Character Recognition using Hebbian Learning Rule

الف)

با تشکیل ماتریس وزن و بازیابی دوباره خروجی ، خروجی جدید را بصورت زیر رسم می‌کنیم :

```
fig,ax=plt.subplots(1,3,figsize=(12,6))
for i,row in enumerate(inputSet.T):
  ax[i].imshow(activation(row,weight).reshape(5,3))
```



شکل 1-1-1 : خروجی بازیابی شده توسط روش Hebbian learning rule

ب)

برای پیدا کردن کوچکترین سایز خروجی که به کمک آن بتوان خروجی را به طور کامل بازیابی کرد، ابتدا به کمک ابزار itertools ، برای تعداد دلخواه خانه موجود در خروجی ، تعداد حالات مختلف bipolar موجود را بدست می‌آوریم . برای مثال برای تعداد 2 خانه برای خروجی ، 4 حالت زیر وجود دارد :

```
list(itertools.product([-1, 1], repeat=2))
```

```
[(-1, -1), (-1, 1), (1, -1), (1, 1)]
```

شکل 1-2-1 : تعداد حالات ممکن برای 3 خانه برای خروجی

همچنین چون در مجموع 3 خروجی داریم ، پس یک انتخاب نیز در اینجا مطرح می‌شود که از بین حالات بالا ، کدام را به خروجی اول ، دوم و سوم نسبت دهیم . به کمک دستور combination برای 4 حالت بالا به تعداد $c(4,3) = 4$ انتخاب داریم :

```
[((-1, -1), (-1, 1), (1, -1)),
 ((-1, -1), (-1, 1), (1, 1)),
 ((-1, -1), (1, -1), (1, 1)),
 ((-1, 1), (1, -1), (1, 1))]
```

شکل 1-2-2 : تعداد انتخاب های ممکن برای انتخاب سه خروجی

با این حساب می‌توان بیان کرد که برای تعداد n خانه ، به تعداد $c(2^n, 3)$ حالت برای انتخاب سه خروجی وجود دارد.(در اینجا n می‌تواند از 2 تا 14 متغیر باشد)

حال تابعی می‌نویسیم(“check_min_size”) که از تعداد خانه 2 شروع به بررسی کرده و تا خانه 14 پیش رود . برای هر تعداد خانه ، حالات انتخاب 3 خروجی را مانند بالا بررسی کرده و برای هر حالت ، ماتریس وزن و در نهایت خروجی بازیابی شده را بدست آورد . در صورتی که خروجی بازیابی شده با خروجی داده شده به شبکه یکسان باشد ، حلقه را متوقف کرده و خروجی را برگرداند.

با اجرای تابع ، برای همان تعداد 2 خانه ، ترکیبی از 3 خروجی پیدا می‌شود که شرط بالا را اغنا می‌کند :

```
Min_out_size=check_min_size(2,14).T
Min_out_size
```

```
The minimal size is 2
array([[-1., -1.,  1.],
       [-1.,  1., -1.]])
```

```
First Minimal output is:
[-1. -1.]

second Minimal output is:
[-1.  1.]

Third Minimal output is:
[ 1. -1.]
```

شکل 1-2-3 : خروجی مطلوب با کمترین سایز ممکن (2 خانه)

پ)

قبل از بررسی حالات مختلف ، دو مفهوم Acc و Tot_Acc را بیان می‌کنیم :

- Acc: در هر بار اجرای الگوریتم ، سه دقت متفاوت برای سه حالت خروجی بدست می‌آید. بنابراین Acc یک ماتریس 100*3 می‌باشد که 100 ، تعداد بار اجرای الگوریتم می‌باشد.
- Tot_Acc: برای هر سطر موجود در ماتریس Acc ، میانگین دقت های بدست آمده برای سه خروجی را به عنوان میانگین کل دقت خروجی بدست می‌آوریم. بنابرین Tot_Acc یک ماتریس 100*1 می‌باشد .

اضافه کردن 10% نویز (خروجی با اندازه 15):

در این حالت ، در هر بار اجرای الگوریتم ، به تعداد 6 عنصر از 63 عنصر ورودی را دارای نویز می‌کنیم و معیار های بالا را برسی می‌کنیم :

```
Acc,Tot_Acc=Test_adding_noise(targetSet,10)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 1-3-1 : دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 10% نویز

اضافه کردن 10% نویز (خروجی با اندازه 2):

```
Acc,Tot_Acc=Test_adding_noise(Min_out_size,10)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 1-3-2 : دقت بدست آمده برای خروجی مینیمال (2 خانه) و اعمال 10% نویز

اضافه کردن 40% نویز (خروجی با اندازه 15):

در این حالت ، در هر بار اجرای الگوریتم ، به تعداد 25 عنصر از 63 عنصر ورودی را دارای نویز می‌کنیم و معیار های بالا را بررسی می‌کنیم :

```
Acc,Tot_Acc=Test_adding_noise(targetSet,40)
```

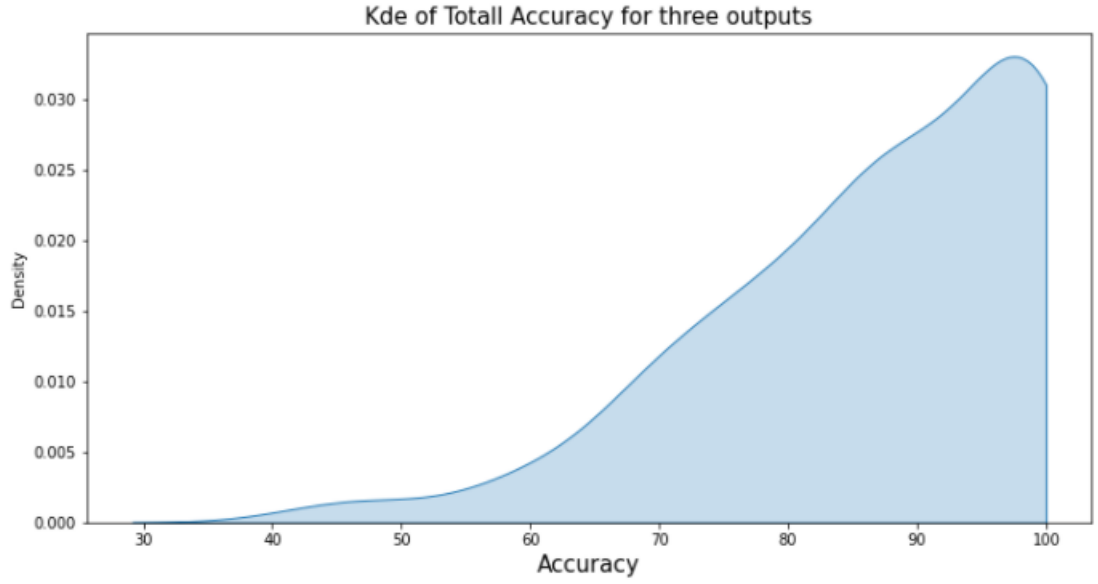
```
Tot_Acc
```

```
array([100.        , 100.        ,  84.44666667, 100.        ,
        84.44666667,  80.00333333, 100.        ,  88.89      ,
        60.        , 100.        ,  95.55666667,  71.11333333,
       100.        ,  77.78      ,  95.55666667,  60.00333333,
        95.55666667,  84.44666667, 100.        ,  95.55666667,
        44.44666667, 100.        ,  68.89      ,  84.44666667,
        48.89      ,  86.67      ,  86.66666667,  91.11333333,
        88.89      ,  88.89      , 100.        ,  95.55666667,
        80.        , 100.        ,  95.55666667,  80.00333333,
        88.89      ,  95.55666667, 100.        ,  77.78      ,
        57.78      ,  88.89      , 100.        , 100.        ,
        77.78      , 100.        ,  71.11333333,  88.89      ,
        88.89      ,  95.55666667,  95.55666667,  80.        ,
       100.        ,  88.89      ,  88.89      ,  86.67      ,
        77.78      ,  80.00333333,  73.33333333,  95.55666667,
       100.        , 100.        ,  68.88666667,  91.11333333,
        68.88666667,  71.11      , 100.        ,  84.44666667,
       100.        , 100.        ,  77.78      , 100.        ,
       100.        ,  88.89      , 100.        ,  84.44666667,
        95.55666667, 100.        ,  80.00333333,  88.89      ,
        84.44666667,  88.89      , 100.        , 100.        ,
        88.89      ,  71.11333333,  68.88666667, 100.        ,
        88.89      , 100.        , 100.        ,  75.55666667,
        86.66666667,  77.78      ,  64.44666667, 100.        ,
        66.67      ,  75.55666667,  73.33333333,  91.11333333])
```

شکل 1-3-3 : دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 40% نویز

در ادامه kde (Kernel density estimation) ، مربوط به Tot_Acc بدست آمده را رسم می‌کنیم :

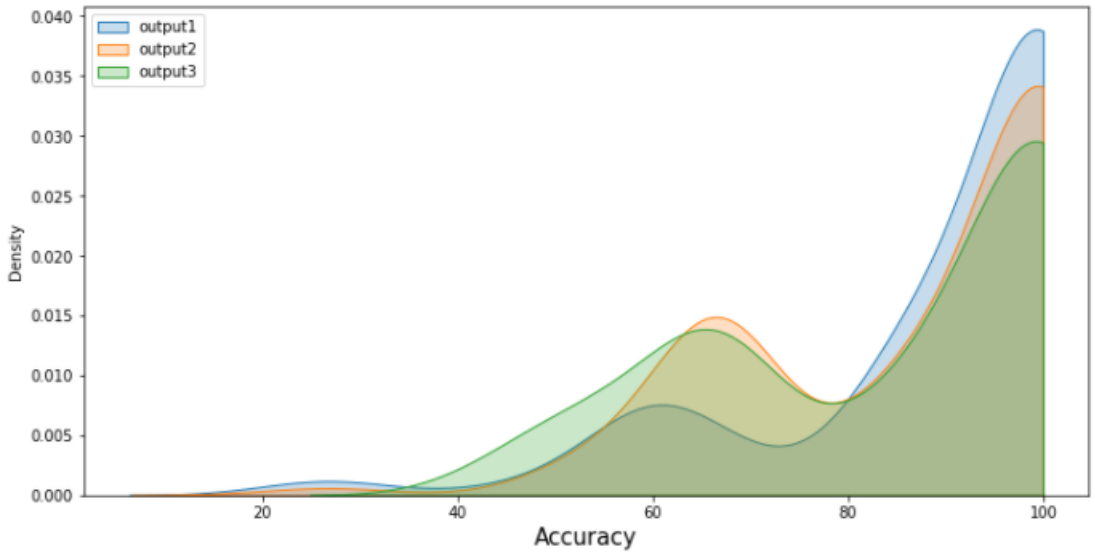
```
we have 30 times 100% totall accuracy
```



شکل 1-3-4 : kde دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 40% نویز

طبق نتایج بدست آمده ، در 30 بار تکرار آزمایش ، هر سه خروجی درست یازیابی شده اند.

همچنین برای هر ستون موجود در Acc (دقت بدست آمده برای هر خروجی) ، kde ها را در یک نمودار رسم می‌کنیم :



شکل 1-3-5 : kde دقت های بدست آمده برای 3 خروجی اصلی (15 خانه) و اعمال 40% نویز

طبق شکل بالا ، برای خروجی اول(پترن A) نسبت به دو خروجی دیگر، بازیابی بهتری داشته ایم .

اضافه کردن 40% نویز (خروجی با اندازه 2):

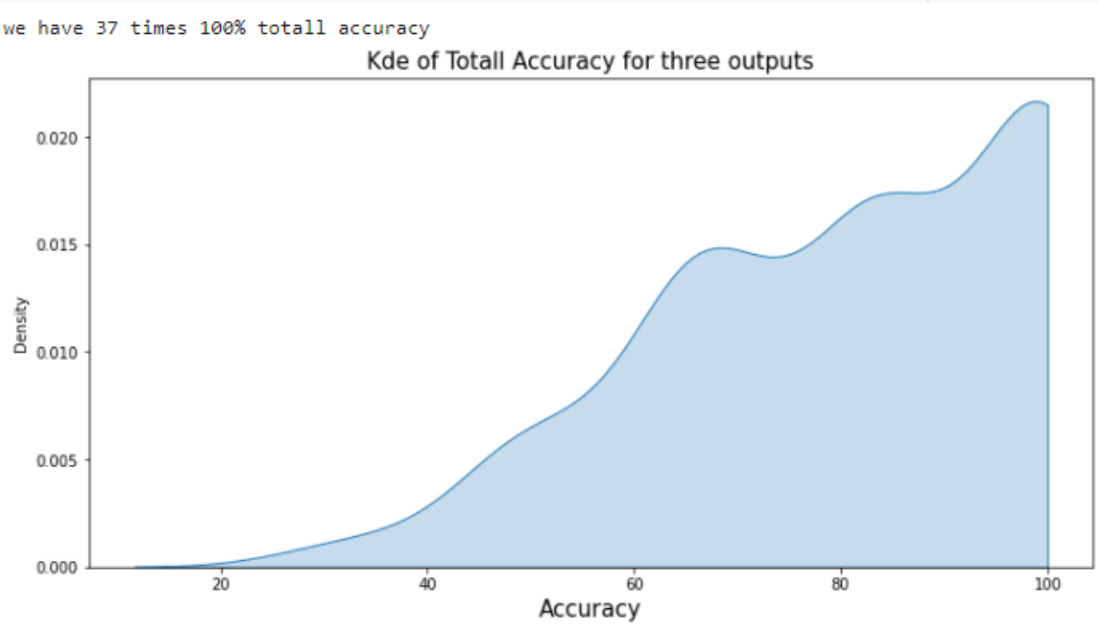
```
Acc,Tot_Acc=Test_adding_noise(Min_out_size,40)
```

```
Tot_Acc
```

```
array([100.        ,  83.33333333,  50.        ,  66.66666667,
        83.33333333, 100.        ,  66.66666667,  83.33333333,
       100.        , 100.        ,  50.        ,  83.33333333,
       100.        ,  66.66666667,  33.33333333, 100.        ,
        66.66666667,  66.66666667, 100.        ,  83.33333333,
       100.        ,  66.66666667,  66.66666667,  66.66666667,
        50.        ,  50.        , 100.        , 100.        ,
       100.        ,  83.33333333,  66.66666667,  83.33333333,
        33.33333333, 100.        , 100.        , 100.        ,
       100.        , 100.        ,  83.33333333,  50.        ,
        66.66666667, 100.        , 100.        ,  66.66666667,
       100.        , 100.        ,  83.33333333,  50.        ,
       100.        , 100.        , 100.        ,  66.66666667,
        50.        ,  83.33333333,  66.66666667,  83.33333333,
        83.33333333,  83.33333333,  66.66666667,  66.66666667,
       100.        ,  66.66666667, 100.        ,  83.33333333,
        83.33333333, 100.        ,  83.33333333,  66.66666667,
        83.33333333,  83.33333333,  83.33333333, 100.        ,
       100.        ,  83.33333333,  83.33333333, 100.        ,
        83.33333333, 100.        , 100.        , 100.        ,
        66.66666667,  50.        ,  66.66666667, 100.        ,
        66.66666667,  83.33333333,  66.66666667,  50.        ,
        83.33333333,  83.33333333, 100.        ,  83.33333333,
        66.66666667, 100.        , 100.        ,  50.        ,
       100.        ,  83.33333333,  66.66666667,  66.66666667])
```

شکل 1-3-6 : دقت بدست آمده برای خروجی مینیمال (2 خانه) و اعمال 40% نویز

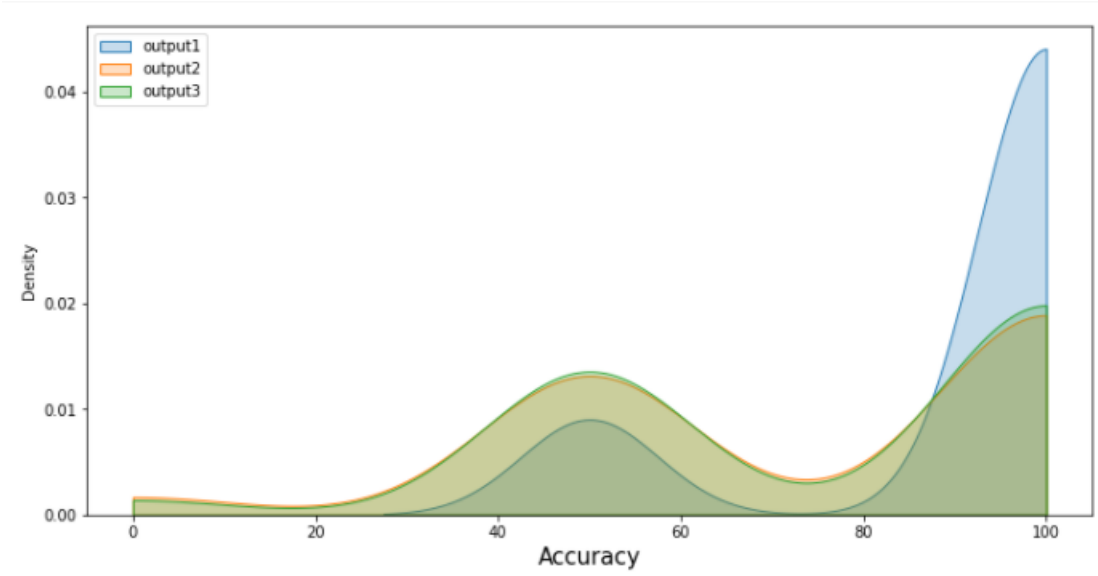
در ادامه kde (Kernel density estimation) ، مربوط به Tot_Acc بدست آمده را رسم می‌کنیم :



شکل 1-3-7 : kde دقت بدست آمده برای خروجی مینیمال (2 خانه) و اعمال 40% نویز

طبق نتایج بدست آمده ، در 37 بار تکرار آزمایش ، هر سه خروجی درست بازیابی شده اند.

همچنین برای هر ستون موجود در Acc (دقت بدست آمده برای هر خروجی) ، kde ها را در یک نمودار رسم می‌کنیم :



شکل 1-3-8 : kde دقت های بدست آمده برای 3 خروجی مینیمال (2 خانه) و اعمال 40% نویز

ت)

اضافه کردن Missing value 10% (خروجی با اندازه 15):

```
Acc,Tot_Acc=Test_adding_Missing(targetSet,10)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 1-4-1 : دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 10% Missing

اضافه کردن Missing value 10% (خروجی با اندازه 2):

```
Acc,Tot_Acc=Test_adding_Missing(Min_out_size,10)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 2-4-1 : دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 10% Missing

طبق نتایج بدست آمده ، با اضافه کردن 10% Missing ، هر دو خروجی اورجینال و مینیمال برای هر 100 تکرار آزمایش درست بازیابی می‌شوند.

اضافه کردن Missing value 40% (خروجی با اندازه 15):

```
Acc,Tot_Acc=Test_adding_Missing(targetSet,40)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 3-4-1 : دقت بدست آمده برای خروجی اصلی (15 خانه) و اعمال 40% Missing

اضافه کردن Missing value 40% (خروجی با اندازه 2):

```
Acc,Tot_Acc=Test_adding_Missing(Min_out_size,40)
```

```
Tot_Acc
```

```
array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
       100.])
```

```
print(f"we have {np.sum((Tot_Acc==100).astype('int'))} times 100% totall accuracy")
```

```
we have 100 times 100% totall accuracy
```

شکل 2-4-1 : دقت بدست آمده برای خروجی مینیمال (2 خانه) و اعمال 40% Missing

همچنین ، با اضافه کردن Missing 40% ، هر دو خروجی اورجینال و مینیمال برای هر 100 تکرار آزمایش درست بازیابی می‌شوند.

د)

طبق نتایج بدست آمده ، مقاومت شبکه در برابر از دست دادن اطلاعات(Missing value) بیشتر از اضافه کردن نویز می‌باشد . (robust تر می‌باشد)

در قسمت قبل هم مشاهد کردیم که برای 40% نویز ، دقت کاهش می‌یابد ولی برای 40% Missing ، همچنان شبکه درست بازیابی می‌کند.

همچنین باتوجه به احتمال ، اگر خروجی با ابعاد کمتری از 15 یافت شود که بتواند خروجی را درست بازیابی کند ، احتمال خطا در خروجی با ابعاد کمتر ، کمتر از خروجی با ابعاد بیشتر خواهد بود .

همینطور که در قسمت پ ، برای خروجی با سایز 15 ، تعداد 30 عدد آزمایش با دقت 100% داشتیم ، درحالیکه این مقدار برای سایز 2 ، 37 عدد بود که با نتیجه بالا همخوانی دارد.

سوال ۲ – Auto-associative Net

1)

به کمک Modified Hebbian rule ، شبکه را آموزش می‌دهیم و ماتریس وزن را تشکیل می‌دهیم . مشخصا ، عناصر روی قطر اصلی ماتریس وزن ، صفر می‌باشد :

```
inputSet = np.hstack((x1,x2))
weight = inputSet @ inputSet.T - inputSet.shape[1]*np.eye(inputSet.shape[0])
weight
```

```
array([[ 0.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 2.,  0.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 2.,  2.,  0.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 2.,  2.,  2.,  0.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0., -2.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.],
       [ 2.,  2.,  2.,  2.,  0.,  0.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  0.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 0.,  0.,  0.,  0., -2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  0.,  2., -2.,  2.,  2.,
         2.,  2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  0., -2.,  2.,  2.,
         2.,  2.],
       [-2., -2., -2., -2.,  0., -2., -2.,  0., -2., -2.,  0., -2., -2.,
        -2., -2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  0.,  2.,
         2.,  2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  0.,
         2.,  2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         0.,  2.],
       [ 2.,  2.,  2.,  2.,  0.,  2.,  2.,  0.,  2.,  2., -2.,  2.,  2.,
         2.,  0.]])
```

```
weight.diagonal()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

شکل 2-1-1 : ماتریس وزن بدست آمده ناشی از دو پترن ورودی

2)

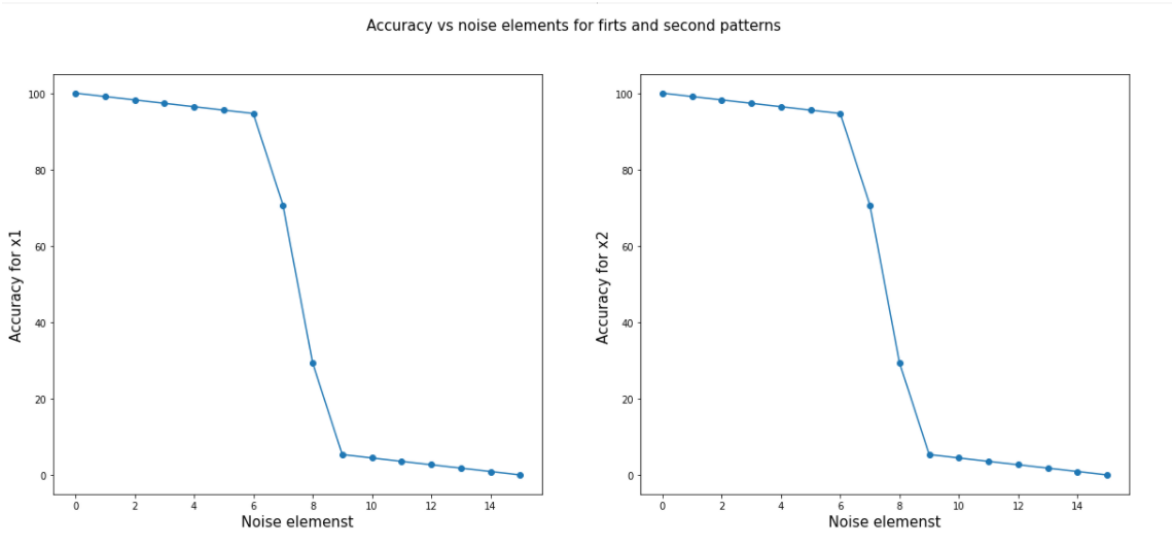
به این منظور ، دو تابع "add_mistake" و "add_missing_value" می‌نویسیم که به ازای تعداد دلخواه داده شده ، در ورودی Missing/Noise ایجاد کند . برای اینکه تمامی حالات ممکن را در نظر بگیریم از

دستور combination موجود در کتابخانه itertools بهره می‌بریم بطوریکه مثلا برای نویزی کردن 6 خانه از 15 خانه ورودی ، تمام $c(15,6)$ حالت را در نظر بگیریم و در یک آرایه ذخیره کنیم .

سپس تابعی بنام "return_acc" می‌نویسیم که با گرفتن تعداد دلخواه Noise/Missing ، برای تمامی حالت های بدست آمده ، پترن بازیابی شده را با پترن اصلی مقایسه می‌کند و میانگین دقت تمامی حالات ($c(15,n)$ که n تعداد دلخواه Noise/Missing می‌باشد) را برگرداند.

*نکته : از آنجایی که ما نمی‌دانیم نویزی کردن یا حذف کردن اطلاعات مروبط به کدام اندیس ورودی (از 1 تا 15) ، تاثیر بیشتری در بازیابی آن دارد ، ترکیب تمامی حالات را در نظر می‌گیریم و سپس بر روی آنها میانگین می‌گیریم .

در زیر نمودار دقت پترن بازیابی شده برای دو پترن ورودی (x1 و x2) ، بر حسب تعداد خانه هایی که دچار Noise شده‌اند ، می‌باشد :

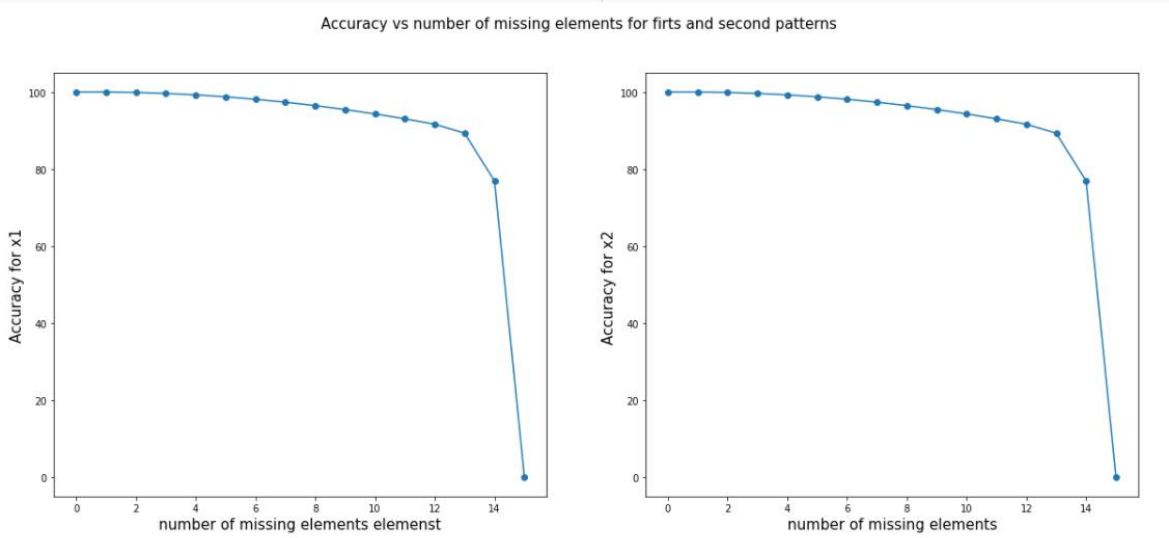


شکل 2-2-1 : دقت پترن بازیابی شده بر حسب تعداد خانه های نویزی در ورودی

طبق نتایج بالا ، با افزودن بیش از 6 نویز (40%) ، دقت به طور چشم گیری کاهش پیدا می‌کند و برای 9 نویز به بالا ، دقت تقریبا صفر می‌باشد . همچنین برای یک اشتباه در ورودی ، دقت تقریبا برابر 100 می‌باشد .

3)

تمامی مراحل فوق را اینبار برای حذف اطلاعات برای 0 تا 15 خانه انجام می‌دهیم و نمودار دقت پترن بازیابی شده برای دو پترن ورودی (x1 و x2) ، بر حسب تعداد خانه هایی که دچار Missing شده‌اند را رسم می‌کنیم :



شکل 2-3-1 : دقت پترن بازیابی شده بر حسب تعداد خانه های Missing در ورودی

طبق نتایج در بالا ، می‌توان گفت که شبکه نسبت به حذف اطلاعات، مقاومتر(robust تر) از افزودن نویز به شبکه می‌باشد . برای مثال در شکل بالا ، با حذف اطلاعات 13 خانه نیز همچنان دقت نسبتا خوبی را دارد ، این در حالی است که در قسمت قبل(افزودن نویز) ، دقت شبکه نزدیک صفر می‌شد.

بنظر می‌رسد ، چون دو پترن ورودی تنها در یک unit (یک خانه) متفاوت می‌باشند ، با نویزی کردن آنها ، شبکه می‌تواند به اشتباه بیفتد . همین شباهت بین دو پترن ورودی باعث می‌شود که شبکه نسبت به حذف اطلاعات robust شود و در صورتی که اطلاعات یکی از دو پترن تا حدی گم شود ، شبکه به کمک پترن دیگر ، پترن اولیه را تا حدی بازیابی می‌کند.

البته اگر به شکل کلی پترن ها نگاه کنیم ، متوجه می‌شویم که در پترن ها در هر جهت تقارن وجود دارد و این خود ایجاب می‌کند که شبکه در حذف اطلاعات نسبت به حذف نویز robust تر باشد.

*مانند این می‌ماند که شما برای یادآوری یک چهارراه ، تنها یک خیابان آن را به خوبی می‌شناسید و با رسیدن به چهار راه و دیدن آن خیابان ، چهار راه را بیاد می‌آورید.

4)

حال اینبار ، ماتریس وزن خود را به کمک پترن به شکل صفر و پترن داده شده توسط سوال(عدد 1) تشکیل می‌دهیم :

```
inputSet_new = np.hstack((x1,x3))
weight_new = inputSet_new @ inputSet_new.T - inputSet_new.shape[1]*np.eye(inputSet_new.shape[0])
weight_new
```

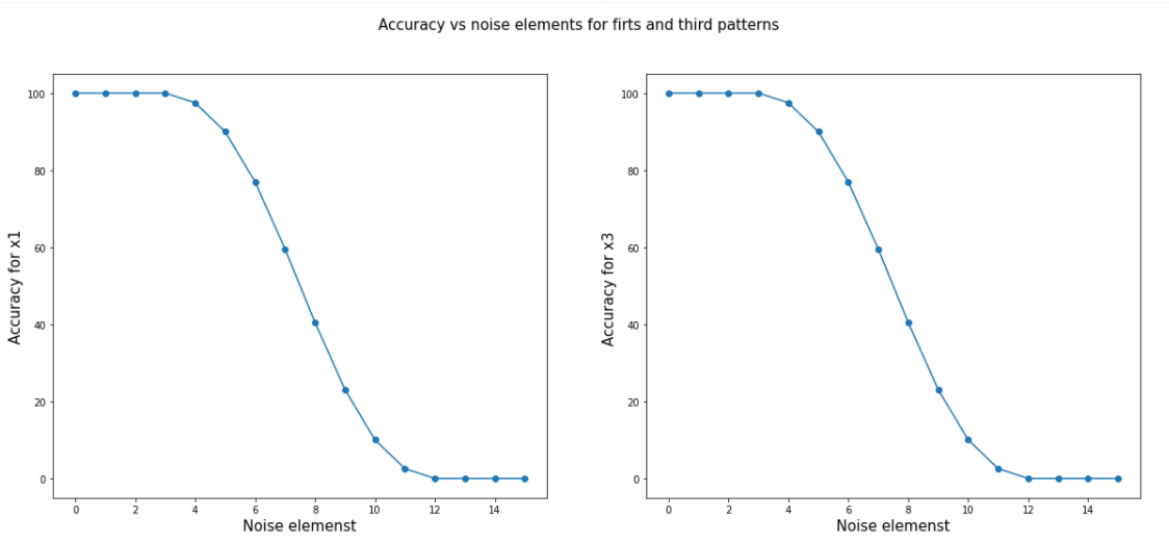
```
array([[ 0.,  0.,  0.,  2., -2.,  0.,  2., -2.,  0.,  2., -2.,  0.,  2.,
         0.,  0.],
       [ 0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,
         2.,  2.],
       [ 0.,  2.,  0.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,
         2.,  2.],
       [ 2.,  0.,  0.,  0., -2.,  0.,  2., -2.,  0.,  2., -2.,  0.,  2.,
         0.,  0.],
       [-2.,  0.,  0., -2.,  0.,  0., -2.,  2.,  0., -2.,  2.,  0., -2.,
         0.,  0.],
       [ 0.,  2.,  2.,  0.,  0.,  0.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,
         2.,  2.],
       [ 2.,  0.,  0.,  2., -2.,  0.,  0., -2.,  0.,  2., -2.,  0.,  2.,
         0.,  0.],
       [-2.,  0.,  0., -2.,  2.,  0., -2.,  0.,  0., -2.,  2.,  0., -2.,
         0.,  0.],
       [ 0.,  2.,  2.,  0.,  0.,  2.,  0.,  0.,  0.,  0.,  0.,  2.,  0.,
         2.,  2.],
       [ 2.,  0.,  0.,  2., -2.,  0.,  2., -2.,  0.,  0., -2.,  0.,  2.,
         0.,  0.],
       [-2.,  0.,  0., -2.,  2.,  0., -2.,  2.,  0., -2.,  0.,  0., -2.,
         0.,  0.],
       [ 0.,  2.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  0.,  0.,
         2.,  2.],
       [ 2.,  0.,  0.,  2., -2.,  0.,  2., -2.,  0.,  2., -2.,  0.,  0.,
         0.,  0.],
       [ 0.,  2.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,
         0.,  2.],
       [ 0.,  2.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,  0.,  2.,  0.,
         2.,  0.]])
```

```
weight_new.diagonal()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

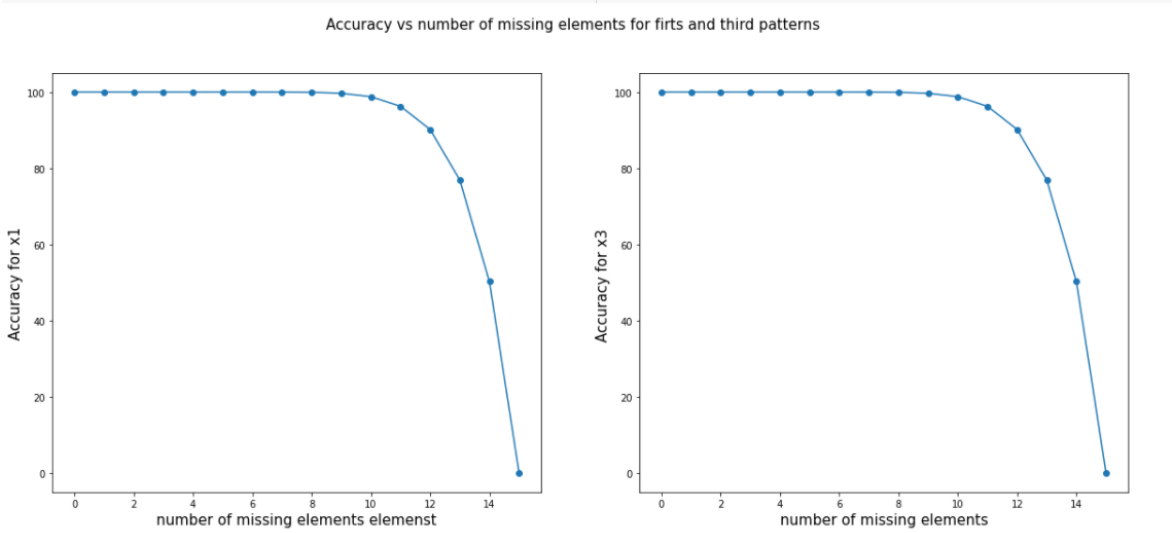
شکل 2-4-1 : ماتریس وزن بدست آمده ناشی از دو پترن جدید ورودی

حال نمودار دقت پترن بازیابی شده برای دو پترن ورودی (x1 و x3) ، بر حسب تعداد خانه هایی که دچار Noise شده‌اند ، می‌باشد :



شکل 2-4-2 : دقت پترن بازیابی شده بر حسب تعداد خانه های نویزی در ورودی

اینبار نمودار دقت پترن بازیابی شده برای دو پترن ورودی (x1 و x3) ، بر حسب تعداد خانه هایی که دچار Missing شده‌اند را رسم می‌کنیم :



شکل 2-4-3 : دقت پترن بازیابی شده بر حسب تعداد خانه های Missing در ورودی

طبق شکل 2-4-2 ، می‌توان گفت هنگامی که دو پترن ورودی تفاوت بیشتری با هم دیگر دارند ، با تعداد خانه های نویزی بیشتری (با افزودن تا 3 نویز) ، همچنان دقت 100% داریم و در واقع افزودن تعداد کمی نویز دقت بهتری را نسبت به حالتی که دو پترن با هم شباهت دارند (قسمت 2) نتیجه می‌دهد .

ولی آیا با افزودن تعداد نویز بیشتر همچنان این نتیجه را می‌توان گرفت ؟

با مقایسه دو شکل 2-4-2 و 2-2-1 می‌توان گفت که وقتی دو پترن شباهت کمتری دارند ، نمودار حالت smooth تری پیدا کرده و در بازه 5 نویز تا 10 نویز حالت خطی به خود می‌گیرد و در خارج این بازه شیب آن آهسته تر کاهش یا افزایش می‌باشد . در حالیکه وقتی دو ورودی شباهت دارند ، نمودار در بازه 6 تا 9 نویز حالت خطی داشته ولی شیب آن بیشتر است و در خارج این بازه به سرعت کاهش یا افزایش می‌یابد .

*در واقع ما با ایجاد تفاوت بین دو پترن ورودی ، حساسیت شبکه را نسبت به نویز کاهش دادیم و شبکه نسبت به نویز مقاوم تر خواهد بود . بنابراین روند تغییرات دقت ، روند مناسب تری خواهد بود .

همچنین نتیجه جالب دیگر در تحلیل حذف اطلاعات می‌باشد . با مقایسه دو نمودار 2-4-3 و 2-3-1 ، می‌توان نتیجه گرفت هنگامی که دو پترن ورودی شباهت بیشتری به هم دارند ، با حذف اطلاعات کمتری همچنان می‌توان دقت قابل قبولی گرفت (در نمودار 2-4-3 تا حذف 10 واحد از اطلاعات

همچنان دقتی نزدیک به 100% داریم درحالیکه در نمودار 2-3-1 تا حذف 3 واحد از اطلاعات ، دقت 100% داریم).

که با تحلیل آورده شده در انتهای بخش 3 ، قابل توجیه است .

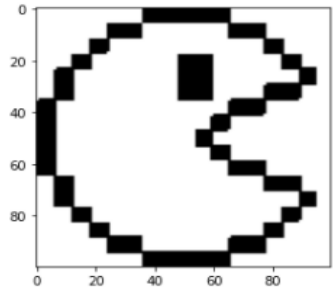
همچنین مانند قسمت قبل ، با ایجاد تفاوت در بین دو ورودی ، نمودار حالت smooth تری به خود گرفته و نسبت به حذف اطلاعات ، حساسیت کمتری خواهد داشت .

سوال 3 – Discrete Hopfield Net

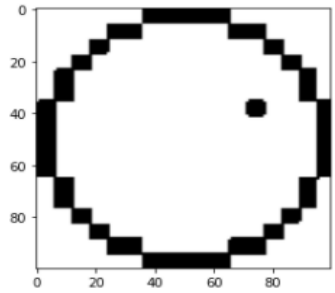
1)

با اجرای کد قرار داده شده ، به تصویر آموزش و تست زیر می‌رسیم :

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



شکل 3-1-1 : شکل های سیاه و سفید باینری آموزش و تست برای عکس pacman

2)

ماتریس وزن را به کمک تک عکس ورودی داده شده توسط ماتریس x (ماتریس ورودی) می‌سازیم :

```
x_reshape=x.reshape((x.shape[0]**2,1))
weight=x_reshape @ x_reshape.T-np.eye(x_reshape.shape[0])
weight
```

```
array([[0., 1., 1., ..., 1., 1., 1.],
       [1., 0., 1., ..., 1., 1., 1.],
       [1., 1., 0., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 0., 1., 1.],
       [1., 1., 1., ..., 1., 0., 1.],
       [1., 1., 1., ..., 1., 1., 0.]])
```

```
np.unique(weight)
```

```
array([-1.,  0.,  1.])
```

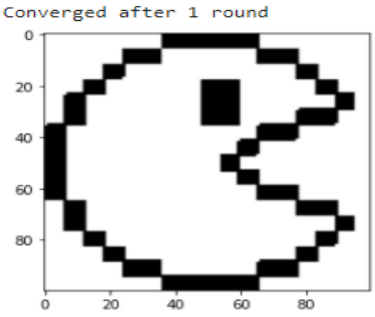
شکل 3-2-1 : ماتریس وزن ایجاد شده توسط ماتریس ورودی x

3)

کد مربوط به این قسمت را به دو بخش زیر تقسیم می‌کنیم :

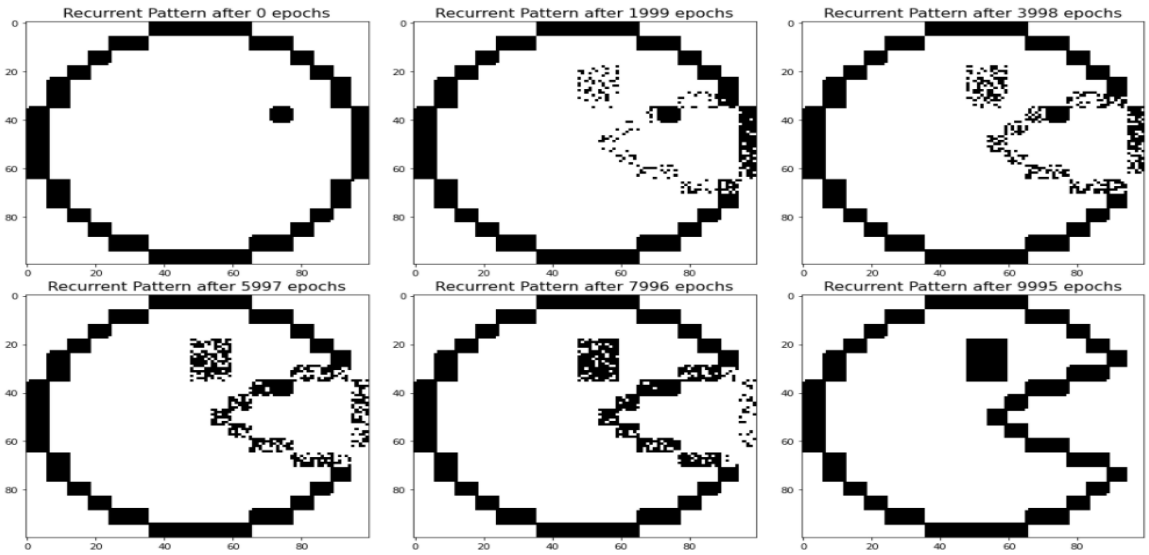
- قسمت اول : در این قسمت تابعی می‌نویسیم که در یک دور کامل (1 round) ، برای تمام 100^2 خانه موجود در ورودی ، طبق قاعده Discrete Hopfiled Net ، پترن اولیه را بازیابی کند و در انتهای هر دور ، پترن بازیابی شده را با پترن اولیه مقایسه کند و اگر این دو یکسان بودند ، توقف کند .

طبق این الگوریتم با دادن تصویر سمت راست برای یک دور(10000 epochs) تصویر بازیابی شده بر تصویر اصلی منطبق بوده و متوقف می‌شویم :



شکل 3-3-1 : تصویر همگرا شده بعد از یک دور بروز رسانی تمام 10000 unit

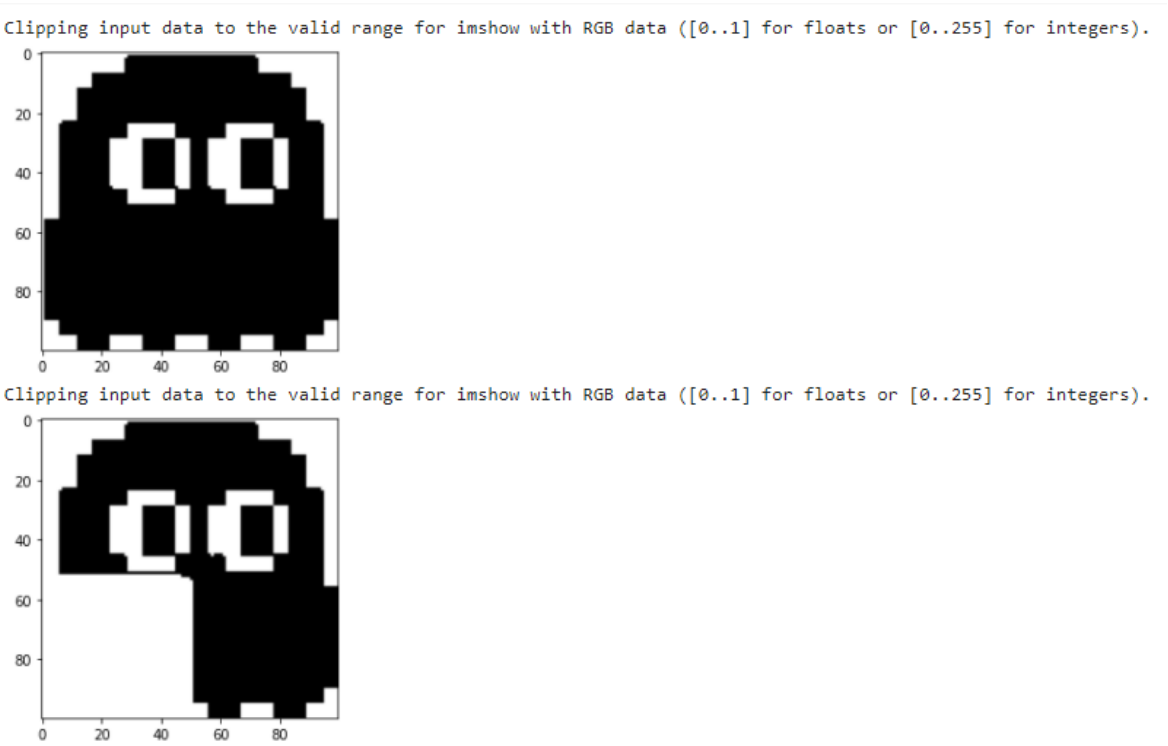
- قسمت دوم : در این قسمت طبق خواسته سوال ، به ترتیب طی هر epoch ، unit به unit شکل داده شده را بروز رسانی می‌کنیم و در طی epoch های مشخص ، خروجی بدست آمده را رسم کرده و همگرایی آنرا برسی می‌کنیم :



شکل 3-3-2 : روند تغییرات عکس بازیابی شده در طی epoch های مختلف

4)

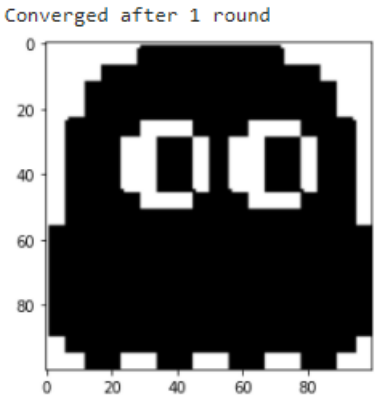
در ابتدا به دلخواه ، قسمتی از عکس داده شده را crop می‌کنیم و به عنوان عکس تست به شبکه در نظر می‌گیریم . دو عکس باینری آموزش و تست به صورت زیر می‌باشد :



شکل 3-4-1 : شکل های سیاه و سفید باینری آموزش و تست برای عکس ghost

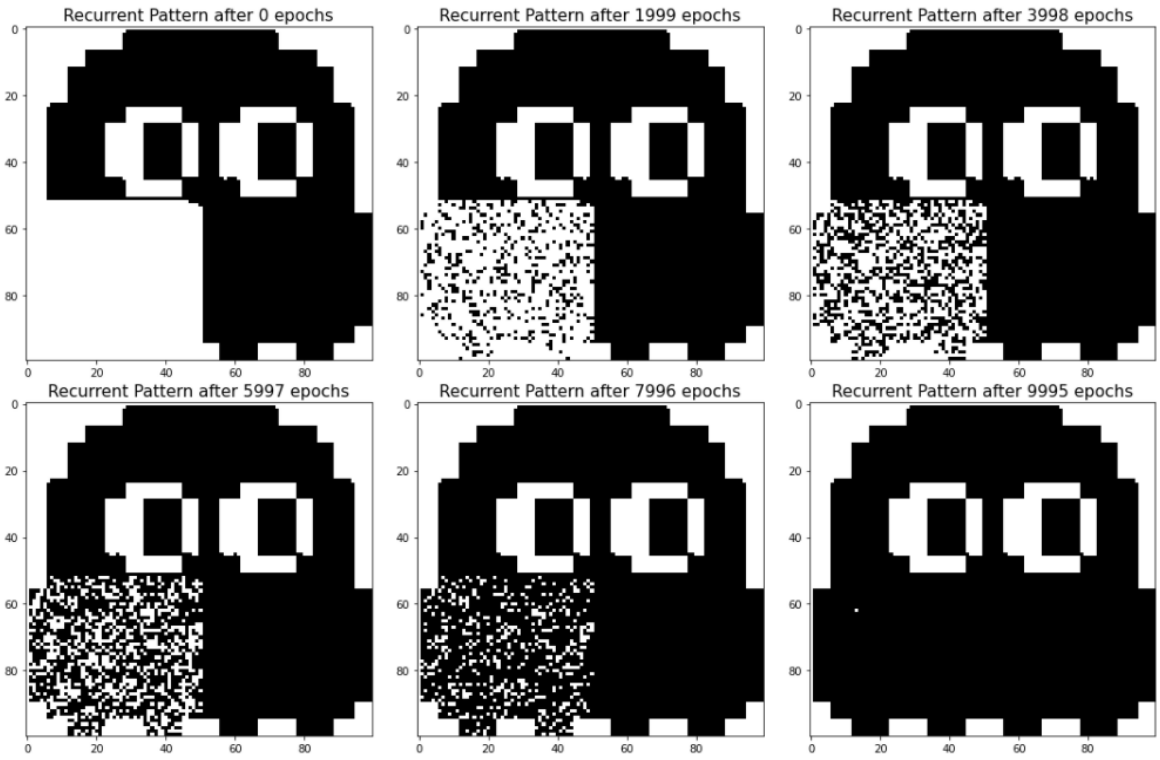
در ادامه ، دو قسمت ذکر شده در بخش 3 را تکرار می‌کنیم :

- قسمت اول :



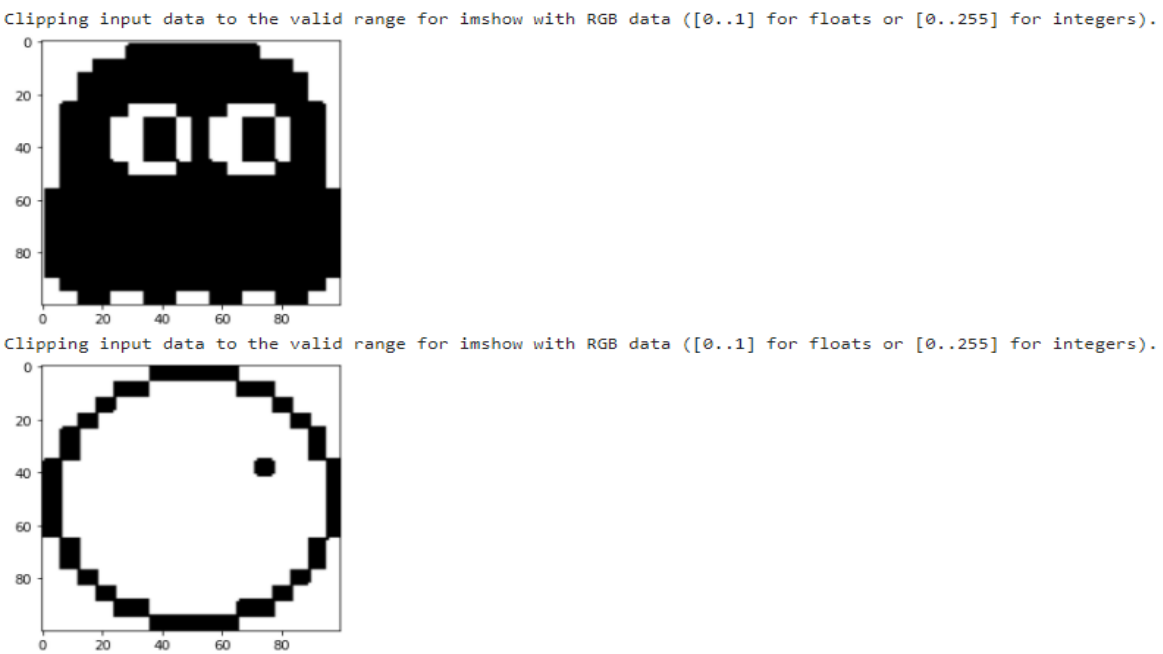
شکل 3-4-2 : تصویر همگرا شده بعد از یک دور بروز رسانی تمام unit 10000

قسمت دوم



شکل 3-4-3 : روند تغییرات عکس بازیابی شده در طی epoch های مختلف

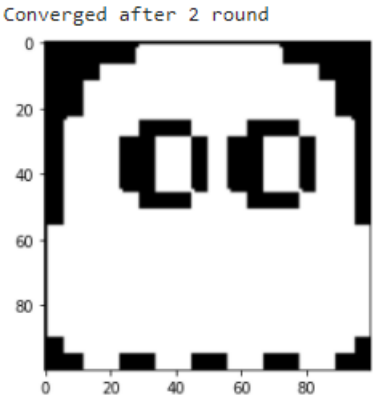
در نهایت طبق خواسته سوال ، پترن ghost را ه عنوان پترن آموزش و پترن pacman_test را به عنوان پترن تست در نظر می‌گیریم و در زیر رسم می‌کنیم :



شکل 3-4-4 : شکل های سیاه و سفید باینری آموزش و تست برای عکس های ghost , pacman_test

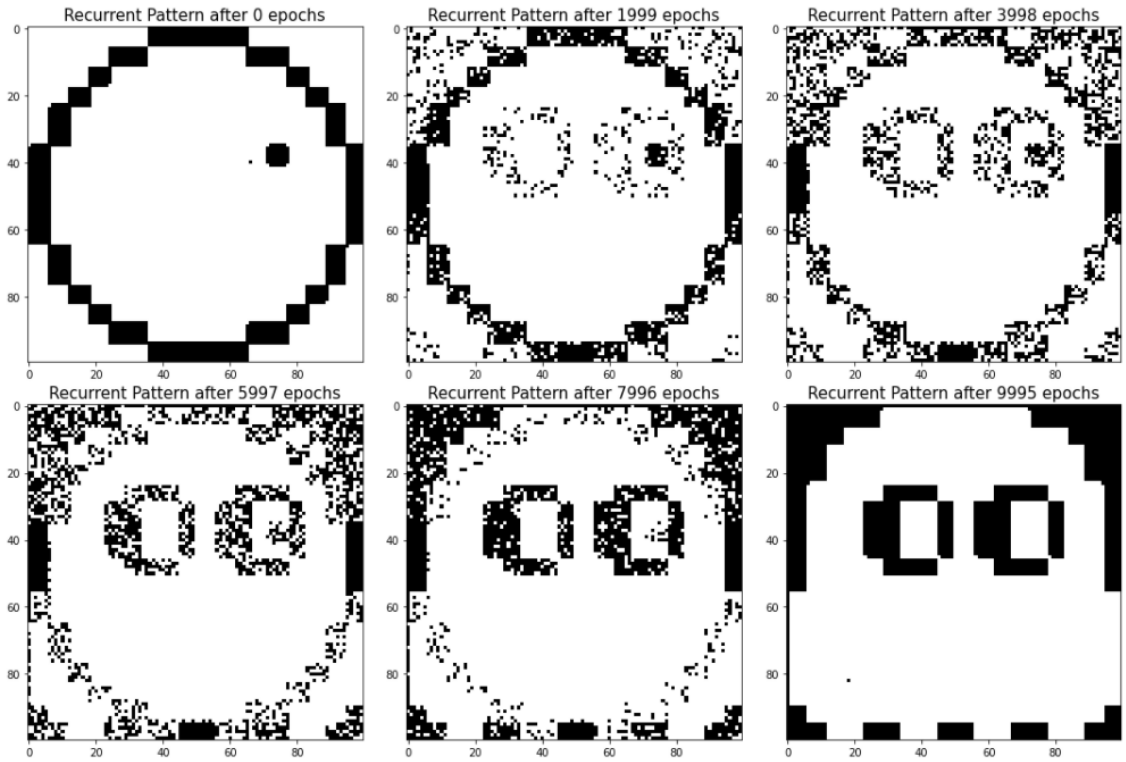
در ادامه ، دو قسمت ذکر شده در بخش 3 را تکرار می‌کنیم :

- قسمت اول :



شکل 3-4-5 : تصویر همگرا شده بعد از دو دور بروز رسانی تمام 10000 unit

همانطور که مشاهده می‌شود ، شکل بازیابی شده به شکل اصلی همگرا نشده و بنابراین تا دو round پیش می‌رویم و به دلیل اینکه عکس بدست آمده با عکس بازیابی شده در اولین round تفاوتی ندارد ، متوقف می‌شویم :



شکل 3-4-6 : روند تغییرات عکس بازیابی شده در طی epoch های مختلف

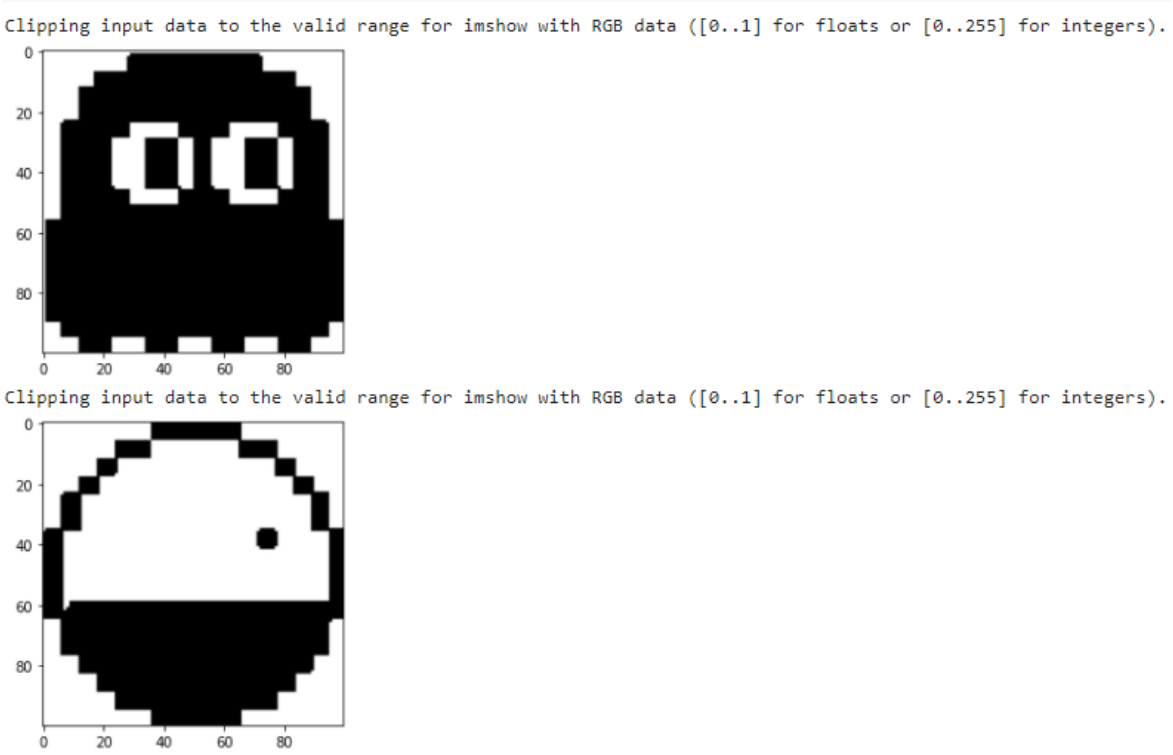
طبق نتایج بالا ، طبیعتا چون عکس تست داده شده به شبکه از جنس ورودی نیست ، شبکه در بازیابی آن دچار مشکل شده است .

اما برای بررسی دلیل همگرایی به خروجی بدست آمده در شکل 3-4-5 ، باید توجه داشت که در بروز رسانی unit به unit الگوریتم Hofild ، unit منتاظر ورودی وجود دارد .

از آنجا که مقادیر موجود در ماتریس ورودی شبکه (1 و 1-) در بسیاری از خانه ها، مکمل مقادیر موجود در ماتریس تست داده شده به شبکه می‌باشد (با مقایسه خانه های متناظر در دو عکس آموزش و تست ، می‌توان دید که در اکثر اوقات ، منتاظر با یک خانه سیاه در عکس ورودی ، یک خانه سفید در عکس تست وجود دارد و برعکس)

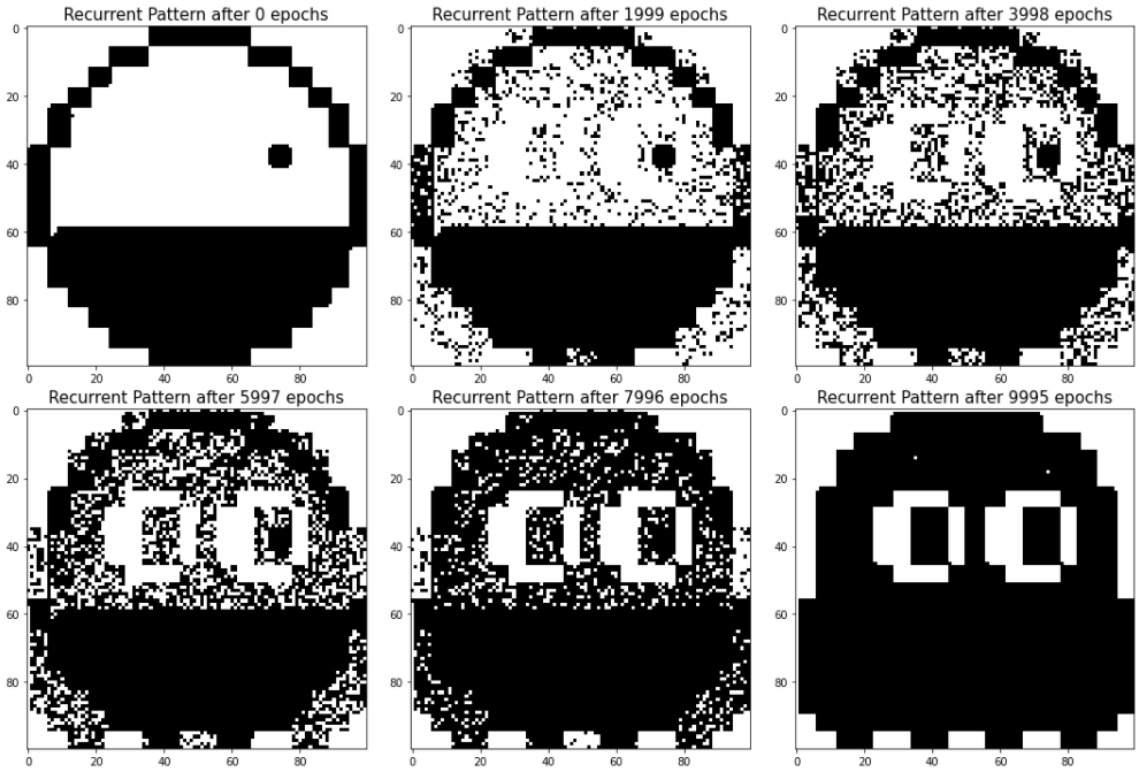
بنابراین ما به شبکه عکسی با درصد زیاد نویزی از ورودی آموزش داده شده به آن را نشان می‌دهیم (در اکثر unit های منتظر ، 1 به جای 1- و 1- به جای 1 وجود دارد) و چیزی که شبکه بازیابی می‌کند ورژن کاملا نویزی شده ورودی خود می‌باشد .

برای بررسی ادعای بالا ، مقدار از درصد نویز موجود در عکس تست داده شده به شبکه را کاهش می‌دهیم :



شکل 3-4-7 : شکل های سیاه و سفید باینری آموزش و تست برای عکس های ghost , pacman_test1

حال نتایج برای epoch های مختلف رسم می‌کنیم :



شکل 3-4-8 : روند تغییرات عکس بازیابی شده در طی epoch های مختلف

همانطور که مشاهده می‌شود ، ادعای بالا درست بود و گرفتن مقداری از نویز عکس تست ، می‌توان همچنان عکس ورودی را بازیابی کرد.

سوال 4 – Bidirectional Associative Memory

الف)

در ابتدا ، ماتریس های ورودی ، خروجی و وزن مربوط به سه پترن اول را در زیر می‌آوریم :

```
three_seq
```

```
array([[-1,  1,  1],
       [ 1,  1,  1],
       [ 1,  1,  1],
       [ 1,  1,  1],
       [-1, -1, -1],
       [-1, -1,  1],
       [ 1,  1,  1],
       [-1,  1,  1],
       [-1, -1,  1],
       [ 1,  1,  1],
       [-1, -1,  1],
       [-1, -1, -1],
       [-1,  1,  1],
       [ 1,  1, -1],
       [ 1,  1,  1]])
```

```
three_target
```

```
array([[-1, -1, -1],
       [-1, -1,  1],
       [-1,  1, -1]])
```

```
three_seq=np.hstack((x1,x2,x3))
three_target=np.hstack((y1,y2,y3)).T
# Calculate weight Matrix for firt three sample: W
W=three_seq @ three_target.T
W
```

```
array([[-1,  1,  1],
       [-3, -1, -1],
       [-3, -1, -1],
       [-3, -1, -1],
       [ 3,  1,  1],
       [ 1,  3, -1],
       [-3, -1, -1],
       [-1,  1,  1],
       [ 1,  3, -1],
       [-3, -1, -1],
       [ 1,  3, -1],
       [ 3,  1,  1],
       [-1,  1,  1],
       [-1, -3,  1],
       [-3, -1, -1]])
```

شکل 4-1-1 : ماتریس ورودی ، خروجی و ماتریس وزن

ب)

برای این منظور دو تابع به نام ’TestInputs‘ و ’TestTargets‘ می‌نویسیم که در اولی خروجی حاصل از ورودی داده شده را توسط ماتریس وزن محاسبه می‌کنیم و در دومی ورودی بازیابی شده توسط خروجی را توسط ماتریس وزن محاسبه می‌کنیم :

```
def testInputs(y_old , x, weight):
  # Multiply the input pattern with the weight matrix
  # (weight.T X x)
  y = weight.T @ x
  for i,row in enumerate(y.T):
    row[row < 0] = -1
    row[row >= 0] = 1
    row[row==0] =y_old.T[i,:][np.where(row==0)[0]]
  return np.array(y)
```

شکل 4-2-1 : خروجی بازیابی شده بعد از بدست آوردن ماتریس ورودی

```
def testTargets(x_old, y, weight):
  # Multiply the target pattern with the weight matrix
  # (weight X y)
  x = weight @ y
  for i,row in enumerate(x.T):
    row[row < 0] = -1
    row[row > 0] = 1
    row[row==0] =x_old.T[i,:][np.where(row==0)[0]]
  return np.array(x)
```

شکل 4-2-2 : ورودی بازیابی شده بعد از بدست آوردن ماتریس خروجی

همانطور که در دو تابع نوشته شده مشاهده می‌شود ، قبل از ورود به تابع فعال ساز ، در صورتی یکی از unit ها برابر صفر باشد ، مقدار متناظر با اندیس آن در ورودی / خروجی قبلی جایگزین می‌کنیم .

در ادامه در قالب یک while loop ، شرط توقف را چک می‌کنیم و تا جایی که مقدار قبلی خروجی با مقدار جدید خروجی بدست آمده (همینطور برای ورودی) برابر نباشد (همگرا نشده باشیم) ادامه می‌دهیم . باید توجه کرد که شرط فوق به معنای همگرا شدن به ورودی و خروجی مطلوب شبکه نیست و به این معنا است که شبکه بیشتر از این قابلیت یادآوری خروجی ها و Pattern های در ورودی را ندارد .

خروجی های حاصل از اجرای الگوریتم را در زیر می‌بینیم :

```
number of iteration is 1

Reconstructed y1:

[-1 -1 -1]


Reconstructed y2:

[-1 -1 -1]


Reconstructed y3:

[-1 -1 -1]


element by element comparison of the given output and the reconstructed one :
 [[ True  True  True]
 [ True  True False]
 [ True False  True]]
```

شکل 4-2-3 : خروجی بازیابی شده و مقایسه آن با خروجی اصلی در اولین epoch

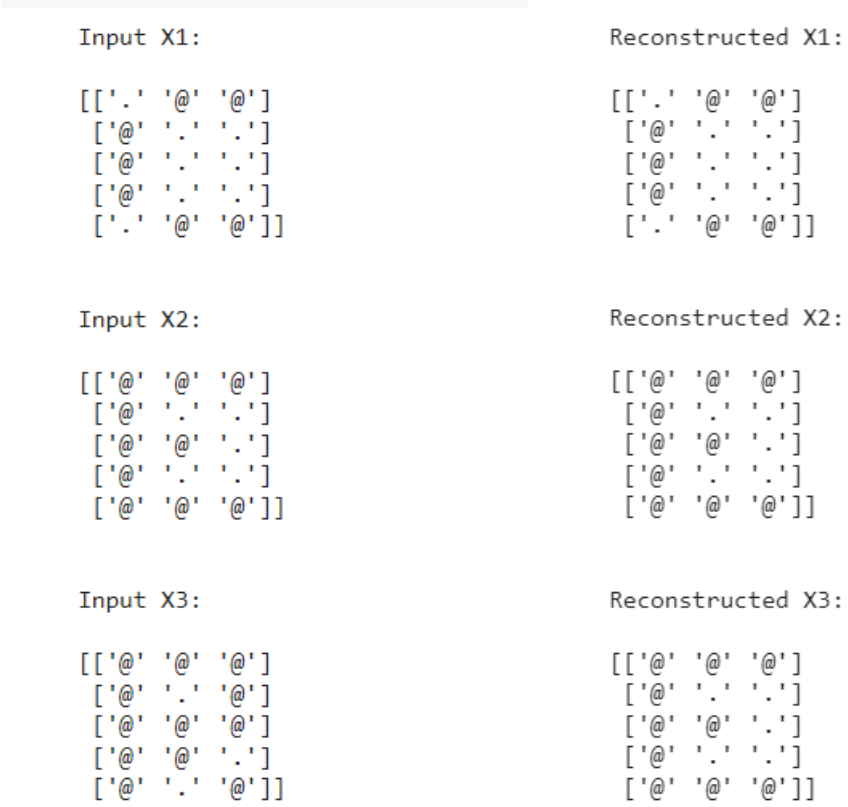
```
Input X1:

[['.' '@' '@']
 ['@' '.' '.']
 ['@' '.' '.']
 ['@' '.' '.']
 ['.' '@' '@']]

Input X2:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]

Input X3:

[['@' '@' '@']
 ['@' '.' '@']
 ['@' '@' '@']
 ['@' '@' '.']
 ['@' '.' '@']]
```

```
Reconstructed X1:

[['.' '@' '@']
 ['@' '.' '.']
 ['@' '.' '.']
 ['@' '.' '.']
 ['.' '@' '@']]

Reconstructed X2:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]

Reconstructed X3:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]
```

شکل 4-2-4 : ورودی بازیابی شده و مقایسه آن با ورودی اصلی در اولین epoch

```
number of iteration is 2

Reconstructed y1:

[-1 -1 -1]


Reconstructed y2:

[-1 -1 -1]


Reconstructed y3:

[-1 -1 -1]


element by element comparison of the given output and the reconstructed one :
 [[ True  True  True]
 [ True  True False]
 [ True False  True]]
```

شکل 4-2-5 : خروجی بازیابی شده و مقایسه آن با خروجی اصلی در دومین epoch



شکل 4-2-6 : ورودی بازیابی شده و مقایسه آن با ورودی اصلی در دومین epoch

طبق نتایج بالا ، در بین سه خروجی داده شده ، شبکه تنها اولین خروجی را درست به یاد می‌آورد و در دو خروجی دیگر ، یک unit را اشتباه به یاد می‌آورد .

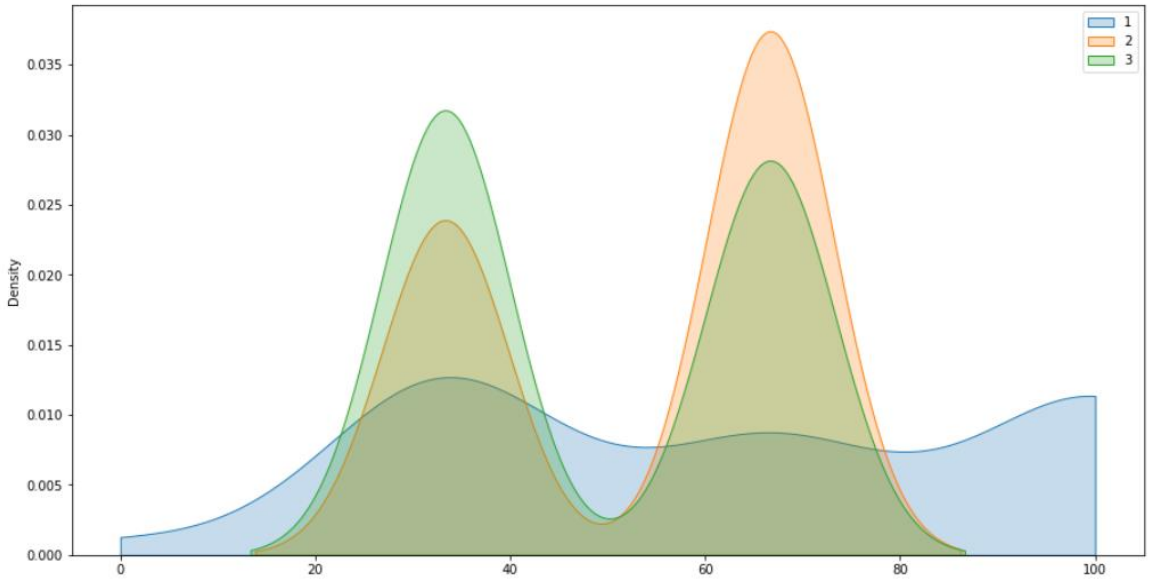
همچنین در بین ورودی ها (Patterns) ، دو ورودی اول (C و E) به درستی بازیابی می‌شوند ولی ورودی سوم (R) ، به حرف E همگرا می‌شود و شبکه در به یاد آوری این پترن به مشکل می‌خورد .

همچنین در تعداد 2 epoch ، شبکه همگرا می‌شود .

پ)

در این قسمت به طور رندوم از بین 15 unit ورودی ، 6 ورودی (40%) را انتخاب کرده و علامت آنها را تغییر می‌دهیم (از 1+ به 1- و بر عکس) و خروجی را تغییر نمی‌دهیم و تمام مراحل قسمت قبل را برای 100 بار تکرار آزمایش انجام می‌دهیم .

در زیر kde (kernel density estimation) مربوط به درصد موفقیت حدس هر unit از 3 unit موجود در خروجی را برای هر پترن (اگر خروجی مربوط به یک پترن درست بازیابی شود ، درصد موفقیت 100% است.) نشان می‌دهد :



شکل 4-3-1 : kdeplot مربوط به خروجی بازیابی شده مربوط به هر پترن

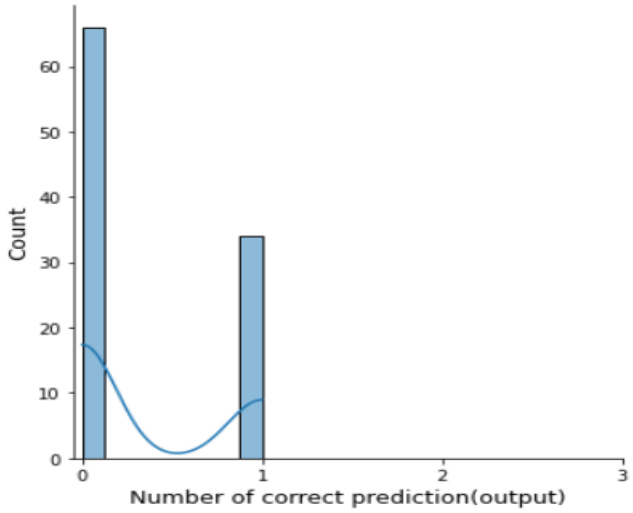
طبق شکل بدست آمده ، بعد از تکرار 100 بار آزمایش ، خروجی مربوط به پترن دوم و سوم تقریبا جمع دو گوسی با میانگین های 33 (یکی از 3 تا درست) و 66 (2 تا از 3 تا درست) می‌باشد و با نویزی کردن پترن های دوم و سوم ، مدل در بازیابی خروجی به اشتباه می‌افتد و در نهایت 2 تا از 3 unit خروجی را درست به یاد می‌آورد .

همچنین طبق شکل برای میانگین 66 ، احتمال یادآوری خروجی دوم بیشتر از خروجی سوم می‌باشد (نمودار نارنجی بالاتر از نمودار سبز رنگ است) . که طبق نتایج بدست آمده در بخش قبل ، (پترن سوم در بازیابی به پترن دوم همگرا می‌شد) قابل استناد است .

ولی برای نمودار آبی رنگ (خروجی اول) ، یک نمودار نسبتا شبیه به توزیع uniform داریم و تقریبا میشود گفت که در هر بار نویزی کردن پترن اول ، خروجی مربوط به آن به احتمال 1/3 بازیابی می‌شود.

(به احتمال 1/3 ، هر سه unit مربوط به خروجی اول درست به یادآورده می‌شوند) .

همچنین count plot زیر ، تعداد خروجی درست بازیابی شده از 3 خروجی موجود (یک خروجی درست بازیابی شده ، اگر هر 3 unit مربوط به آن درست بدست آمده باشد) را نشان می‌دهد :



شکل 4-3-2 : Count plot مربوط به تعداد خروجی درست بازیابی شده از 3 خروجی موجود

طبق نمودار بالا ، با نویزی کردن ورودی ، تقریبا در 65 حالت ، هر سه خروجی اشتباه به یادآورده می‌شوند و در 35 حالت ، تنها یک خروجی از 3 خروجی (خروجی مربوط به پترن اول (C)) درست بازیابی می‌شوند .

ت)

طبق نتیجه بدست آمده از قسمت ب ، با دادن دو پترن اول (C و E) به شبکه ، در طی epoch های محدود ، همان پترن ها به یادآورده می‌شوند . پس ماکسیمم دو پترن می‌توان به شبکه داد .

ث)

مراحل ب و ت را برای تمام 6 پترن ورودی تکرار می‌کنیم :

```
number of iteration is 1

Reconstructed y1:

[-1 -1 -1]


Reconstructed y2:

[-1 -1 -1]


Reconstructed y3:

[-1 -1 -1]


Reconstructed y4:

[-1  1  1]


Reconstructed y5:

[ 1 -1 -1]


Reconstructed y6:

[ 1 -1  1]


element by element comparison of the given output and the reconstructed one :
 [[ True  True  True  True  True  True]
 [ True  True False  True  True  True]
 [ True False  True  True  True  True]]
```

شکل 4-4-1 : خروجی بازیابی شده و مقایسه آن با خروجی اصلی در اولین epoch

```
Input X1:              Reconstructed X1:      Input X4:              Reconstructed X4:

[['.' '@' '@']         [['.' '@' '@']         [['.' '@' '.']         [['.' '@' '.']
 ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '@']          ['@' '.' '@']
 ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '@']          ['@' '.' '@']
 ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '@']          ['@' '@' '@']
 ['.' '@' '@']]         ['.' '@' '@']]         ['.' '@' '.']]         ['.' '@' '@']]

Input X2:              Reconstructed X2:      Input X5:              Reconstructed X5:

[['@' '@' '@']         [['@' '@' '@']         [['@' '@' '@']         [['@' '@' '@']
 ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '.']
 ['@' '@' '.']          ['@' '@' '.']          ['@' '@' '.']          ['@' '@' '.']
 ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '.']
 ['@' '@' '@']]         ['@' '@' '@']]         ['@' '.' '.']]         ['@' '.' '.']]

Input X3:              Reconstructed X3:      Input X6:              Reconstructed X6:

[['@' '@' '@']         [['@' '@' '@']         [['@' '@' '@']         [['@' '@' '@']
 ['@' '.' '@']          ['@' '.' '.']          ['@' '.' '@']          ['@' '.' '.']
 ['@' '@' '@']          ['@' '@' '.']          ['@' '@' '@']          ['@' '@' '.']
 ['@' '@' '.']          ['@' '.' '.']          ['@' '.' '.']          ['@' '.' '.']
 ['@' '.' '@']]         ['@' '@' '@']]         ['@' '.' '.']]         ['@' '.' '.']]
```

شکل 4-4-2 : ورودی بازیابی شده و مقایسه آن با ورودی اصلی در اولین epoch

```
number of iteration is 2

Reconstructed y1:

[-1 -1 -1]


Reconstructed y2:

[-1 -1 -1]


Reconstructed y3:

[-1 -1 -1]


Reconstructed y4:

[-1  1  1]


Reconstructed y5:

[ 1 -1 -1]


Reconstructed y6:

[ 1 -1 -1]


element by element comparison of the given output and the reconstructed one :
 [[ True  True  True  True  True  True]
 [ True  True False  True  True  True]
 [ True False  True  True  True False]]
```

شکل 4-4-3 : خروجی بازیابی شده و مقایسه آن با خروجی اصلی در دومین epoch

```
Input X1:

[['.' '@' '@']
 ['@' '.' '.']
 ['@' '.' '.']
 ['@' '.' '.']
 ['.' '@' '@']]

Reconstructed X1:

[['.' '@' '@']
 ['@' '.' '.']
 ['@' '.' '.']
 ['@' '.' '.']
 ['.' '@' '@']]

Input X4:

[['.' '@' '.']
 ['@' '.' '@']
 ['@' '.' '@']
 ['@' '.' '@']
 ['.' '@' '.']]

Reconstructed X4:

[['.' '@' '.']
 ['@' '.' '@']
 ['@' '.' '@']
 ['@' '@' '@']
 ['.' '@' '@']]

Input X2:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]

Reconstructed X2:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]

Input X5:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '.' '.']]

Reconstructed X5:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '.' '.']]

Input X3:

[['@' '@' '@']
 ['@' '.' '@']
 ['@' '@' '@']
 ['@' '@' '.']
 ['@' '.' '@']]

Reconstructed X3:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '@' '@']]

Input X6:

[['@' '@' '@']
 ['@' '.' '@']
 ['@' '@' '@']
 ['@' '.' '.']
 ['@' '.' '.']]

Reconstructed X6:

[['@' '@' '@']
 ['@' '.' '.']
 ['@' '@' '.']
 ['@' '.' '.']
 ['@' '.' '.']]
```

شکل 4-4-4 : ورودی بازیابی شده و مقایسه آن با ورودی اصلی در دومین epoch

در سومین epoch ، نتایج مشابه epoch دوم می‌باشد و به همین دلیل متوقف می‌شویم .

طبق نتایج بدست آمده ، برای 3 پترن اول نتایجی مشابه بخش ب داریم . برای 3 پترن بعدی (O ، F و P) ، پترن F را شبکه کامل به یاد می‌آورد ، پترن O را با دو unit اضافه به یاد می‌آورد و به جای پترن P ، پترن F را به خاطر می‌آورد.

همچنین نکته جالب در مورد بازیابی خروجی در این است که ، در اولین epoch ، تنها خروجی های دوم و سوم (مربوط به پترن های E و R) درست بازیابی نمی‌شوند. اگر به پترن های بدست آمده در اولین epoch نگاه کنید ، متوجه می‌شوید که علاوه بر دو پترن E و R ، شبکه دو پترن F و P را نیز همانند هم به یاد می‌آورد .

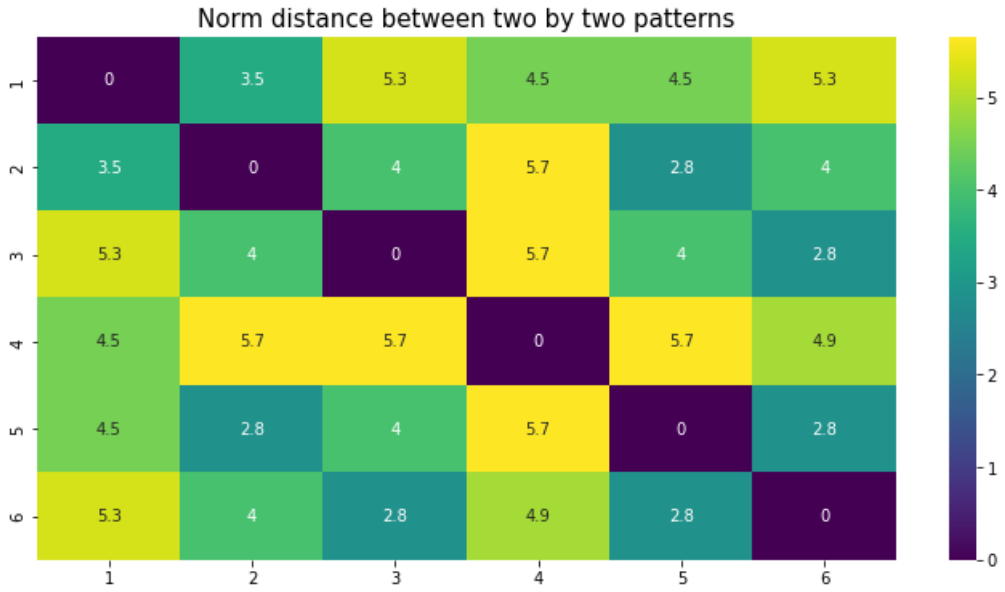
*چون در دومین epoch ، از اطلاعات موجود در epoch اول استفاده می‌شود ، شبکه که در اولین epoch تنها 2 خطا در خروجی داشت ، به دلیل شباهت دو پترن P و F ، در دومین epoch دارای 3 خطا می‌شود . در واقع خروجی مربوط به پترن آخر که در اولین epoch درست بازیابی شده بود در دومین epoch به غلط بازیابی می‌شود .

در نهایت در Epoch سوم ، به دلیل اینکه شبکه قادر به یادآوری بیشتری نمی‌باشد ، متوقف می‌شویم .

بنابراین طبق مشاهدات ، شبکه به جای پترن R ، پترن E ، به جای پترن P ، پترن F و به جای پترن O ، ترکیبی از دو پترن O و C را به یاد می‌آورد .

یکی از دلایل این می‌باشد که شبکه به جای ذخیره کلیات قرار گیری unit ها ، به رابطه بین unit های مشکی و سفید توجه می‌کند ، همانطور که ما برای به یادآوری برخی وقایع ، سعی کنیم ربط آنها را به هم پیدا کنیم .

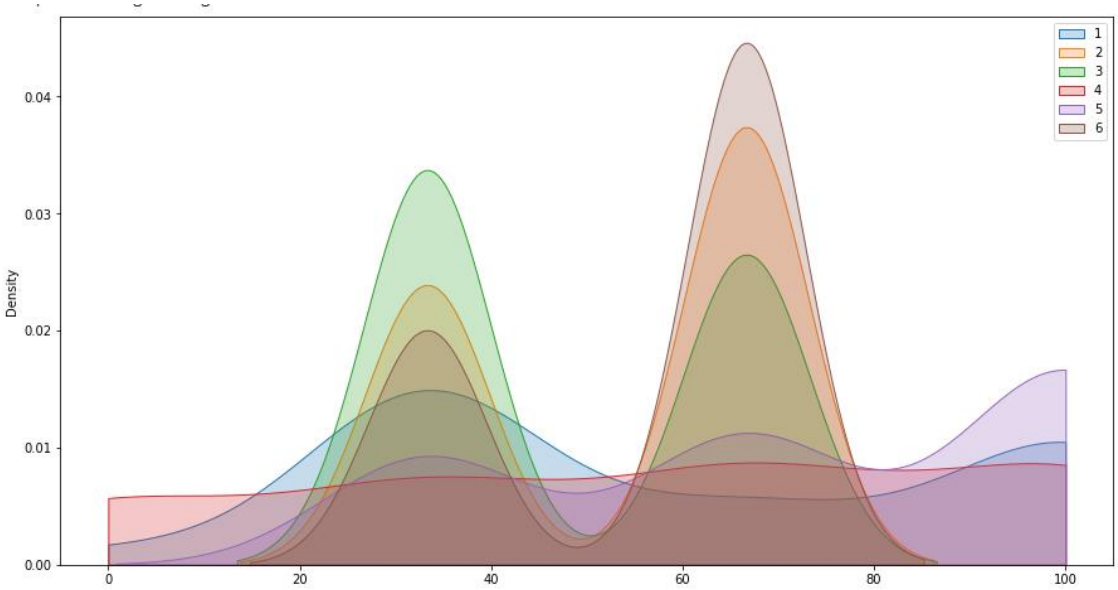
برای اینکه درکی از رابطه بین پترن ها داشته باشیم ، فاصله اقلیدسی دو به دو پترن ها را طبق ماتریس متناظر آنها بدست می‌آوریم و توسط Heat map رسم می‌کنیم :



شکل 4-4-5 : مقایسه دو به دو پترن های ورودی در یک Heat map

طبق شکل بالا هم ، برای مثال پترن های 5 و 6 (F و P) کمترین فاصله (بیشترین شباهت) را بین بقیه پترن ها دارند.

در نهایت مانند قسمت ت ، تعداد 40% ورودی ها را دارای نویز می‌کنیم و اینبار برای هر 6 پترن دو نمودار kde و count را رسم می‌کنیم :



شکل 4-4-6 : kdeplot مربوط به خروجی بازیابی شده مربوط به هر پترن

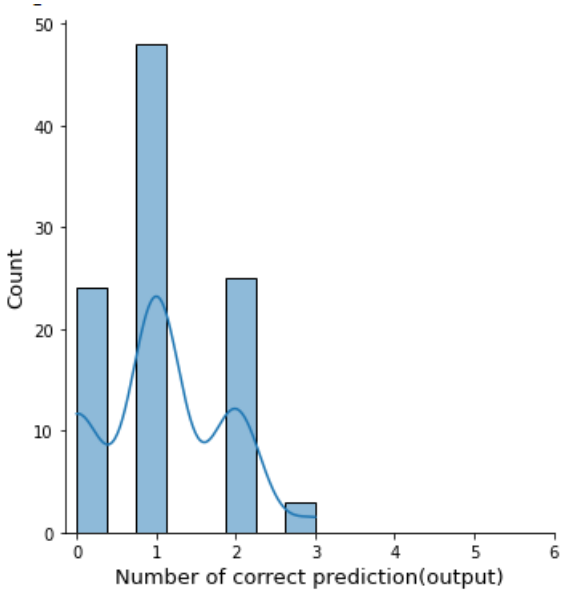
طبق شکل بدست آمده ، بعد از تکرار 100 بار آزمایش ، اینبار خروجی مربوط به پترن دوم و سوم و ششم تقریبا جمع دو گوسی با میانگین های 33 (یکی از 3 تا درست) و 66 (2 تا از 3 تا درست)

می‌باشد و با نویزی کردن پترن های دوم و سوم و ششم ، مدل در بازیابی خروجی به اشتباه می‌افتد و در نهایت 2 تا از 3 unit خروجی را درست به یاد می‌آورد .

همچنین طبق شکل برای میانگین 66 ، احتمال یادآوری خروجی ششم بیشتر از دوم، و دوم بیشتر از خروجی سوم می‌باشد (نمودار قهوه‌ای بالاتر از نارنجی و نارنجی بالاتر از سبز) . که طبق نتایج بدست آمده در بخش قبل ، (پترن سوم در بازیابی به پترن دوم و پترن ششم به پترن پنجم همگرا می‌شد) قابل استناد است .

همچنین برای نمودار نارنجی یک توزیع به تقریب بسیار خوب uniform داریم که نشان می‌دهد احتمال اتفاق هر کدام از حالات 1/3 می‌باشد ولی دو نمودار آبی و بنفش (مربوط به پترن E و F) ، بین دو حالت uniform و گوسی هستند که احتمال وقوع بازیابی درست در خروجی نسبت به نمودار نارنجی بیشتر است . طبق اینکه می‌دانیم پترن های A و F در حالت بدون نویز درست بازیابی می‌شوند این نتیجه منتظره می‌باشد.

همچنین count plot زیر ، تعداد خروجی درست بازیابی شده از 6 خروجی موجود (یک خروجی درست بازیابی شده ، اگر هر 3 unit مربوط به آن درست بدست آمده باشد) را نشان می‌دهد :



شکل 4-4-7 : Count plot مربوط به تعداد خروجی درست بازیابی شده از 6 خروجی موجود

طبق نمودار بالا ، با نویزی کردن ورودی ، تقریبا در 24 حالت ، هر 6 خروجی اشتباه به یادآورده می‌شوند و در 48 حالت ، تنها یک خروجی از 6 خروجی (خروجی مربوط به پترن اول (C)) و در 25 حالت ، دو خروجی از 6 خروجی و در 3 حالت ، سه خروجی از 6 خروجی درست بازیابی می‌شوند .