به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین سری سوم

نام و نام خانوادگی	مریم ریاضی
شماره دانشجویی	۸۱۰۱۹۷۵۱۸
تاریخ ارسال گزارش	1401/2/30

فهرست گزارش سوالات (لطفاً پس از تکمیل گزارش، این فهرست را به‌روز کنید.)

سوال ۱ – Pattern Association using Hebbian Learning Rule

بخش 1:

الگوریتم Hebbian learning rule یک الگوریتم برای شبکه های نورونی حافظه دار است. عملکرد این الگوریتم مطابق گام های زیر است:

1) در ابتدا ماتریسی به ابعاد n*m با مقادیر صفر در نظر می گیریم. این ماتریس، ماتریس وزن ها است. n تعداد المان های ورودی و m تعداد المان های خروجی است.
2) جفت های ورودی و خروجی را به صورت S = [s1, s2, .., sn] و T = [t1, t2, t3, …, tm] در نظر می گیریم و برای هر جفت گام های زیر را انجام می دهیم.
3) $i=1\ldots n\ si\rightarrow xi$
4) $j=1\ldots m\ tj\rightarrow yj$
5) $wij+:=wij-+xiyj$

با توجه به الگوریتم توضیح داده شده می توان نشان داد که ماتریس وزن ها به روش ساده تر و از طریق زیر محاسبه می شود.

$$W = \sum_{p=1}^{P} s(p)t(p)^T$$

بخش 2:

توضیح کد:

در ابتدا داده هایی که به ما داده شده بودند را در کد کپی کردم. این داده ها، شامل سه ورودی و سه خروجی بودند که ورودی ها به صورت آرایه هایی با ابعاد 63*1 و خروجی ها نیز آرایه هایی با ابعاد 15*1 بودند.

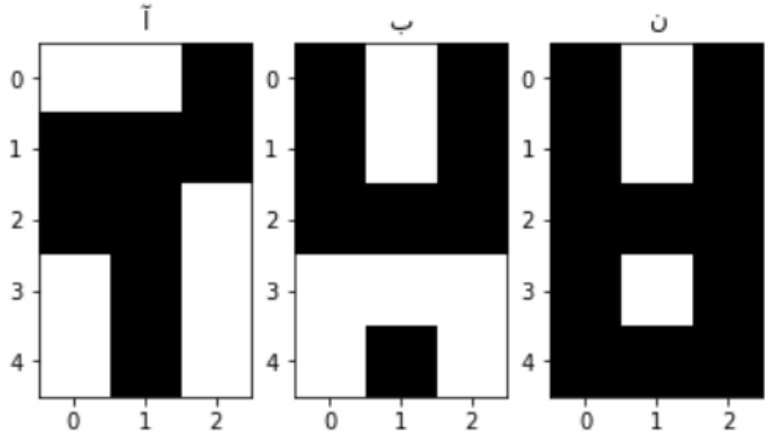
در اولین گام، با استفاده از دستور np.transpose ابعاد آرایه های ورودی و خروجی را به گونه ای تغییر دادم که برای ضرب خارجی و به دست آوردن ماتریس وزن ها مشکلی نداشته باشند. برای محاسبه ماتریس وزن ها، تابعی به نام weight_matrix تعریف شد. در اینجا آرایه های ورودی و خروجی در یکدیگر ضرب خارجی شدند و با یکدیگر جمع شدند.

در ادامه برای کل عملکرد شبکه، تابعی به نام Hebbian تعریف شد. در این تابع ورودی و ماتریس وزن ها با یکدیگر ضرب می شوند و خروجی این ضرب سپس از تابعی به نام Activation عبور داده می شود تا خروجی های اصلی پیدا شوند. خروجی این تابع، آرایه ای به ابعاد 15 است. در ادامه و برای رسم

کردن خروجی ابتدا خروجی این تابع را به صورت ماتریس های 3*5 درآوردم و سپس خروجی ها را رسم کردم.

بخش 3:

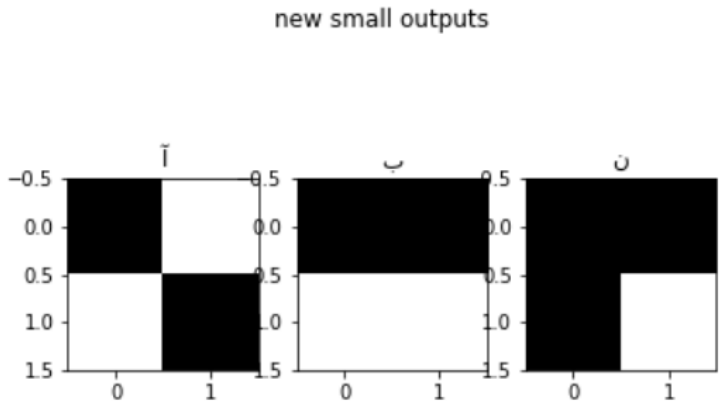
خروجی شبکه را برای ورودی های داده شده رسم کردم و بر اساس شکل های رسم شده می توان نتیجه گرفت که این شبکه به صورت صد در صد درست کار می کند و خروجی ها را به درستی نشان می دهد.



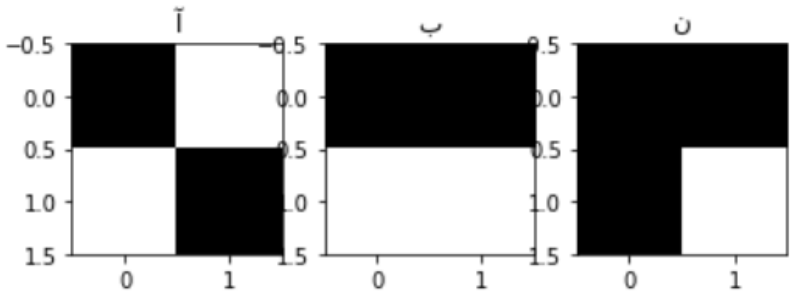
خروجی های شبکه به ازای ورودی های اصلی

بخش 4:

در این بخش برای هر خروجی یک آرایه 4تایی در نظر گرفتم که در واقع بخشی از خروجی ها بودند و برای این خروجی ها ماتریس وزن جدیدی به دست آمد. حالا خود شبکه را با این خروجی ها و ماتریس جدید وزن ها امتحان کردم.



خروجی های کوچک جدید انتخاب شده

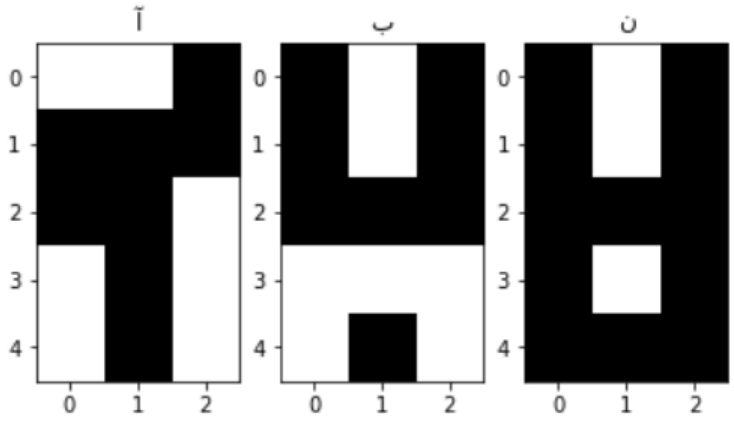


خروجی های شبکه برای ورودی های قبلی و ماتریس وزن جدید

بخش 5:

برای تولید ورودی های نویزی دو تابع تعریف شد. تابع اول، flip نام دارد. کاربرد این تابع این است که از میان المان های رندوم انتخاب شده، مقادیر آن ها را برعکس کند. تابع دوم، noisy_input نام دارد. این تابع در ابتدا ورودی و درصد تغییر را به عنوان ورودی می گیرد و به مقدار دردصد مورد نظر، المان های را از ورودی به صورت تصادفی انتخاب کرده و تابع flip را روی آن ها اجرا می کند.

در ادامه شبکه را مانند قبل و البته با وزن های محاسبه شده از قبل، روی ورودی های نویزی اجرا می کنیم. خروجی ها به ازای 20 و 60 درصد نویز به صورت زیر هستند.

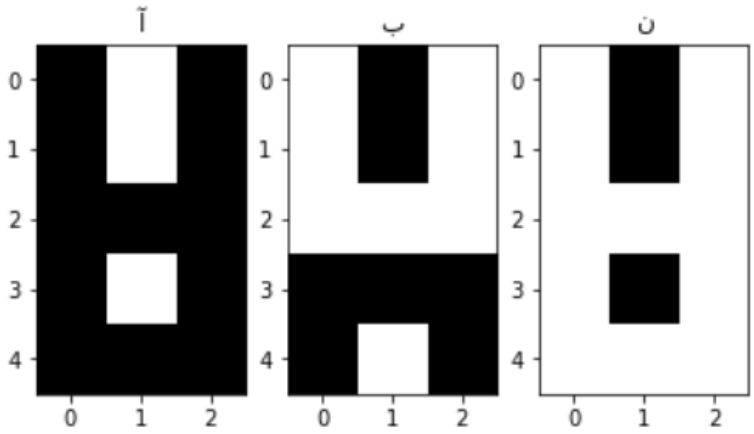


خروجی ها به ازای ورودی هایی با 20 درصد نویز

درصد درستی برای هر حرف به صورت زیر است:

```
Accuracy percentages for original outpus (20% noise):
Accuracy percentage for 20% noise for "آ" is = 99.46666666666665
Accuracy percentage for 20% noise for "ب" is = 100.0
Accuracy percentage for 20% noise for "ن" is = 96.26666666666668
```

درصد درستی خروجی برای هر حرف برای 20 درصد نویز

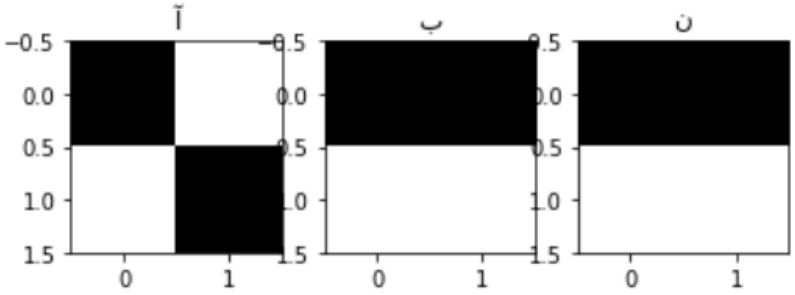


خروجی ها به ازای ورودی هایی با 60 درصد نویز

```
Accuracy percentages for original outpus (60% noise):
Accuracy percentage for 60% noise for "آ" is = 16.266666666666673
Accuracy percentage for 60% noise for "ب" is = 5.866666666666667
Accuracy percentage for 60% noise for "ن" is = 17.333333333333346
```

درصد درستی خروجی برای هر حرف برای 60 درصد نویز

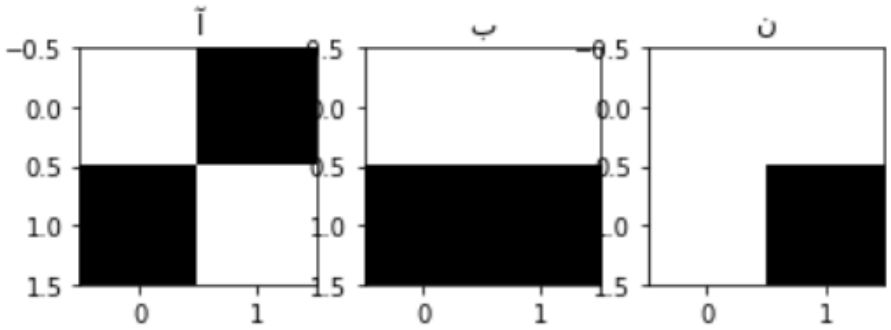
نتایج شبکه برای ورودی نویز دار و خروجی های با ابعاد کوچک:



خروجی شبکه به ازای 20 درصد نویز برای خروجی های کوچک

```
Accuracy percentages for small outpus (20% noise):
Accuracy percentage for 20% noise for "آ" is = 98.5
Accuracy percentage for 20% noise for "ب" is = 100.0
Accuracy percentage for 20% noise for "ن" is = 96.0
```

درصد درستی شبکه برای 20 درصد نویز برای خروجی های کوچک



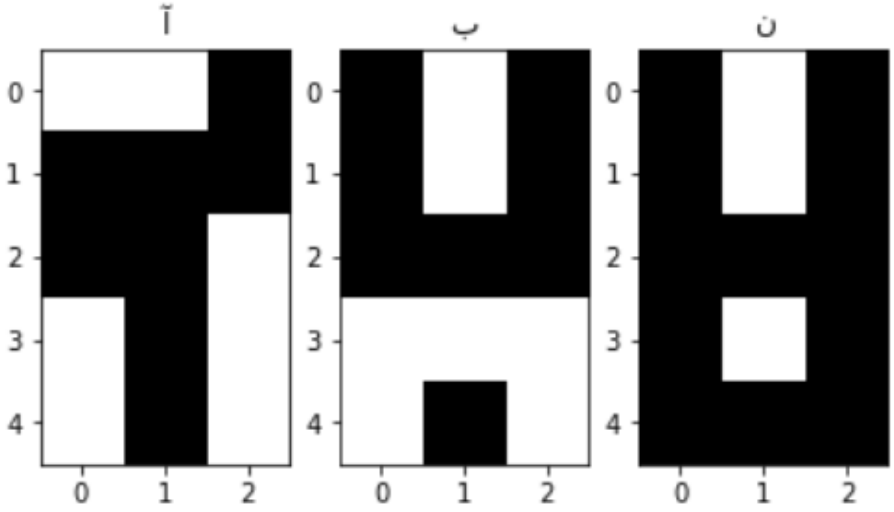
خروجی شبکه به ازای 60 درصد نویز برای خروجی های کوچک

```
Accuracy percentages for small outpus (60% noise):
Accuracy percentage for 60% noise for "آ" is = 18.0
Accuracy percentage for 60% noise for "ب" is = 12.0
Accuracy percentage for 60% noise for "ن" is = 15.25
```

درصد درستی شبکه برای 60 درصد نویز برای خروجی های کوچک

بخش 6:

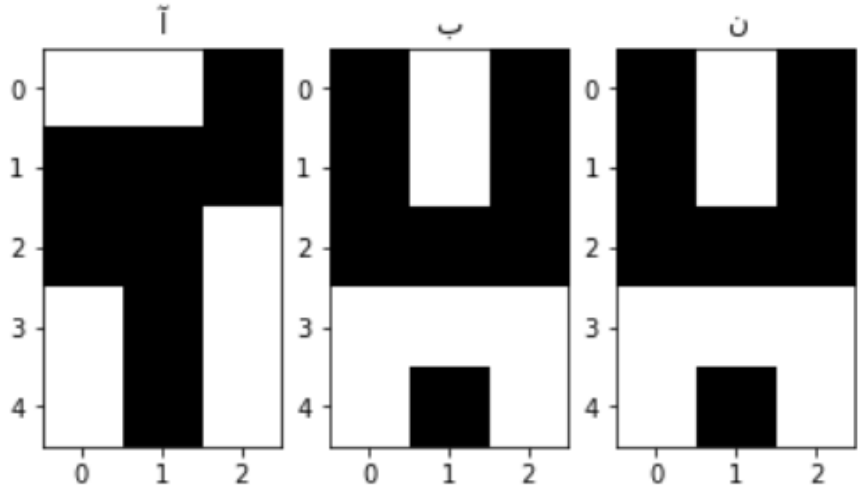
تابع های تعریف شده در این بخش برای تولید ورودی هایی با مقادیر اشتباه نیز مانند بخش قبل هستند با این تفاوت که در این بخش تابعی به نام zero تعریف شده است که مقادیر تصادفی انتخاب شده از ارایه ورودی را به 0 تبدیل می کند.



خروجی ها به ازای ورودی هایی با 20 درصد حذف داده

```
Accuracy percentages for original outpus (20% missing):
Accuracy percentage for 20% missing for "آ" is = 100.0
Accuracy percentage for 20% missing for "ب" is = 100.0
Accuracy percentage for 20% missing for "ن" is = 100.0
```

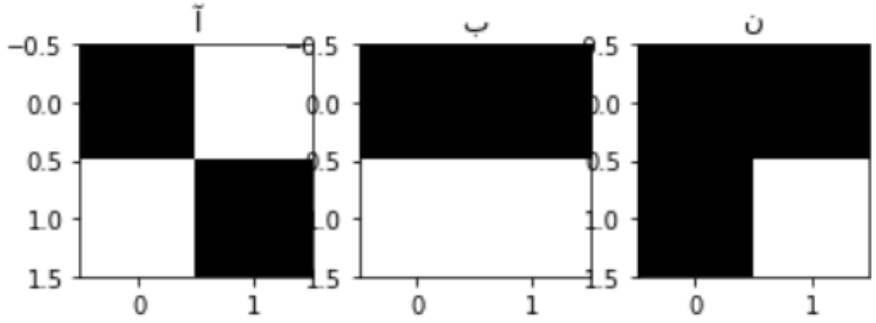
درصد درستی خروجی برای هر حرف برای 20 درصد حذف داده



خروجی ها به ازای ورودی هایی با 60 درصد حذف داده

```
Accuracy percentages for original outpus (60% missing):
Accuracy percentage for 60% missing for "آ" is = 98.8
Accuracy percentage for 60% missing for "ب" is = 100.0
Accuracy percentage for 60% missing for "ن" is = 91.73333333333332
```

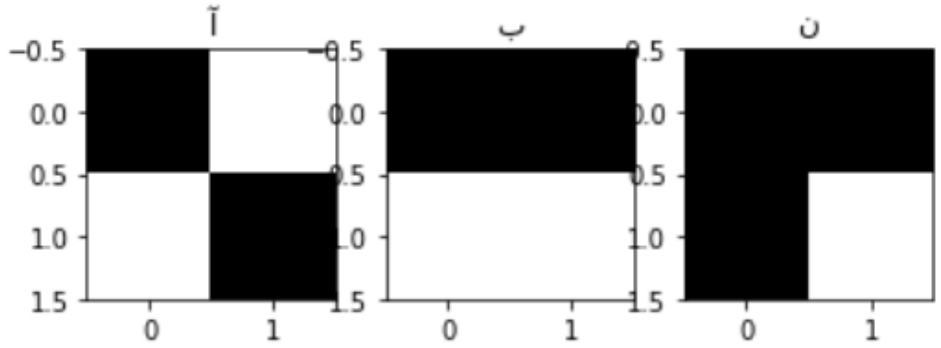
درصد درستی خروجی برای هر حرف برای 60 درصد حذف داده



خروجی شبکه برای 20 درصد داده از دست رفته برای خروجی کوچک

```
Accuracy percentages for small outpus (20% missing):
Accuracy percentage for 20% missing for "آ" is = 100.0
Accuracy percentage for 20% missing for "ب" is = 100.0
Accuracy percentage for 20% missing for "ن" is = 99.75
```

درصد شباهت خروجی شبکه برای 20 درصد داده از دست رفته برای خروجی کوچک



خروجی شبکه برای 60 درصد داده از دست رفته برای خروجی کوچک

```
Accuracy percentages for small outpus (60% missing):
Accuracy percentage for 20% missing for "آ" is = 100.0
Accuracy percentage for 20% missing for "ب" is = 100.0
Accuracy percentage for 20% missing for "ن" is = 100.0
```

درصد شباهت خروجی شبکه برای 60 درصد داده از دست رفته برای خروجی کوچک

سوال ۲ – Auto-associative Net

قسمت 1:

توضیح کد: کد برای این بخش مشابه سوال اول است با این تفاوت که در اینجا و برای Auto associative دیگر خروجی جداگانه نداریم. وزن های شبکه از ضرب کردن ورودی در ترنسپوز شده آن به دست می آید. مطابقه آنچه در درس خواندیم وزن ها برای Hebbian learning rule و Modified hebbian learning rule از قاعده های زیر به دست می آیند.

$$W = \sum_{p=1}^{P} s(p)s(p)^T \qquad W = \sum_{p=1}^{P} s(p)t(p)^T$$

وزن های به دست آمده در کد مطابق شکل زیر هستند:

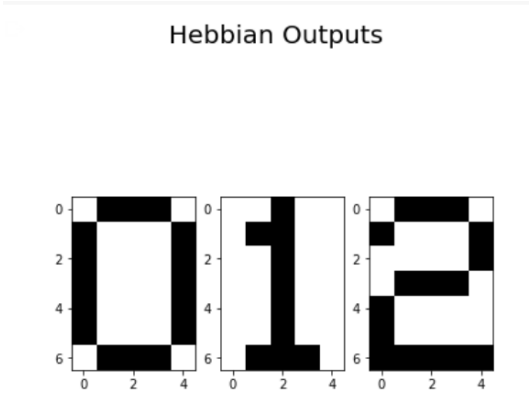
```
Hebbian Weights are:

[[ 3. -1. -3. ... -3. -3.  1.]
 [-1.  3.  1. ...  1.  1.  1.]
 [-3.  1.  3. ...  3.  3. -1.]
 ...
 [-3.  1.  3. ...  3.  3. -1.]
 [-3.  1.  3. ...  3.  3. -1.]
 [ 1.  1. -1. ... -1. -1.  3.]]
Modified Hebbian Weights are:

[[ 0. -1. -3. ... -3. -3.  1.]
 [-1.  0.  1. ...  1.  1.  1.]
 [-3.  1.  0. ...  3.  3. -1.]
 ...
 [-3.  1.  3. ...  0.  3. -1.]
 [-3.  1.  3. ...  3.  0. -1.]
 [ 1.  1. -1. ... -1. -1.  0.]]
```

وزن های به دست آمده برای Hebbian و modified hebbian

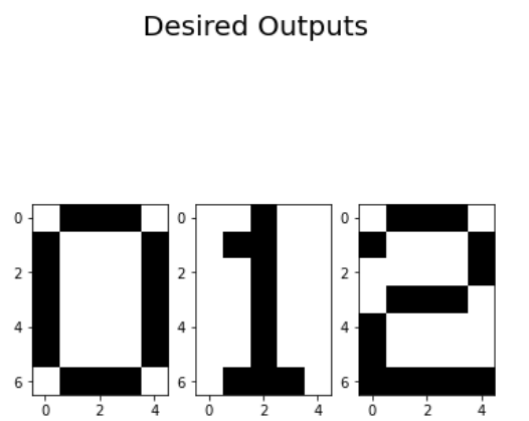
قسمت 2:

وزن های به دست آمده برای Hebian را روی ورودی های شبکه اعمال می کنیم.

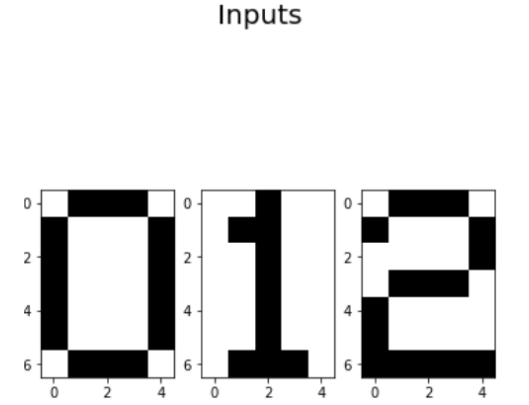
نتایج:



خروجی های شبکه



خروجی های مطلوب



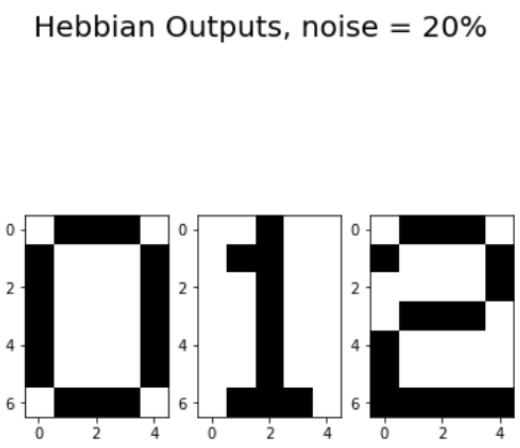
ورودی های اولیه

با توجه به شکل های گذاشته شده می بینیم که خروجی شبکه و خروجی مطلوب کاملا مطابق یکدیگر هستند و شبکه توانسته است عملکرد خوبی داشته باشد.

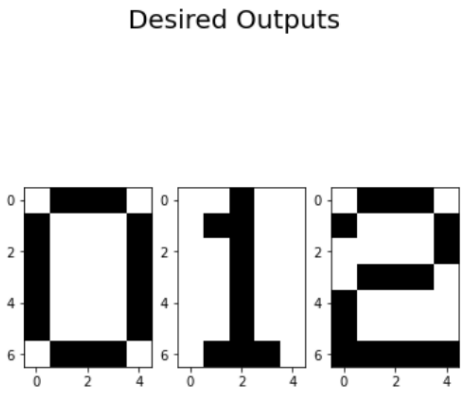
قسمت 3:

در این قسمت برای اعمال نویز از تابع های سوال 1 استفاده شده است.

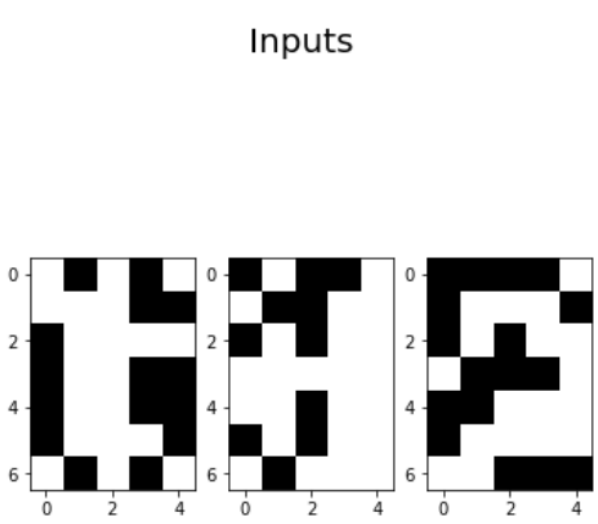
نویز 20 درصد:



خروجی های شبکه

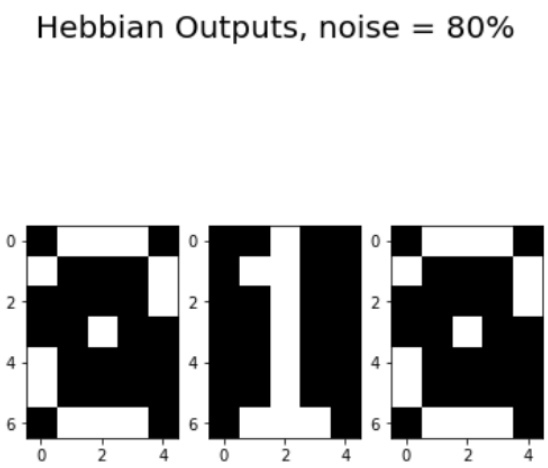


خروجی های مطلوب

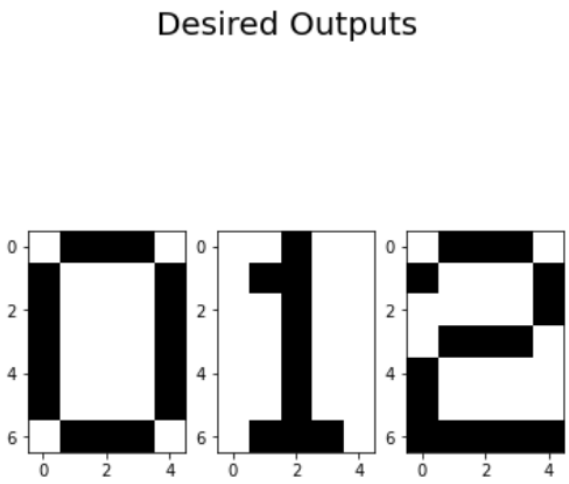


ورودی های نویزی

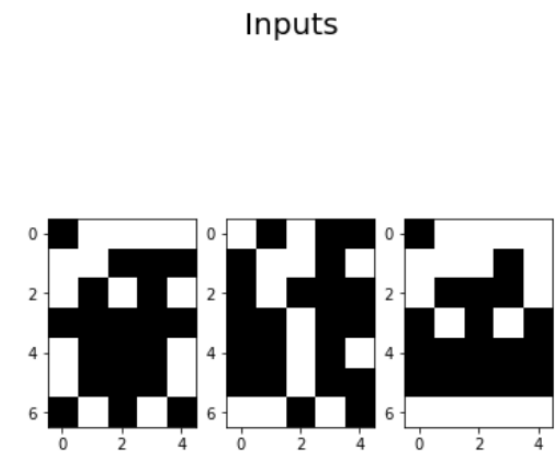
نویز 80 درصد:



خروجی های شبکه



خروجی های مطلوب



ورودی های نویزی

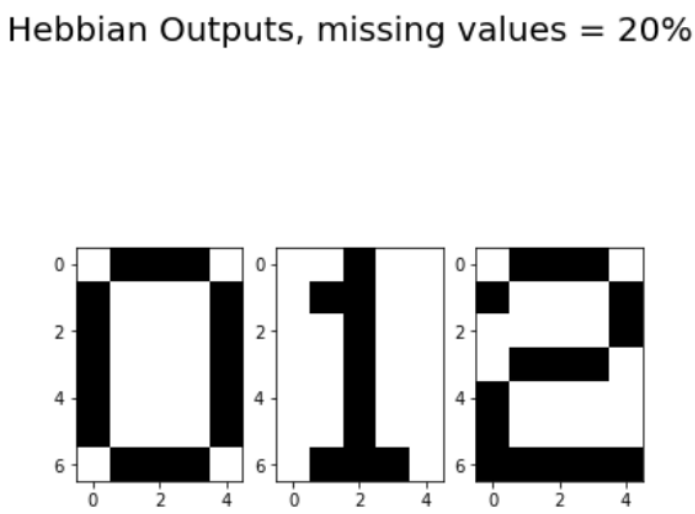
الف) با توجه به شکل نتایج اولا میفهمیم که شبکه در مقابل نویز 20 درصد مقاوم است و توانسته از این ورودی های نویزی به خروجی مطلوب برسد. اما در نویز 80 درصد دیگر مقاومت نداشته و خروجی های کاملا اشتباهی حاصل می شوند.

ب) در اعمال نویز یک الگوریتم یکسان نسبت به اعداد 1- ,1 اعمال می شود که در واقع جابجا کردن آن ها است. با توجه به اینکه رفتار یکسان و یکنواختی نسبت به این دو عدد صورت گرفته می توان نتیجه گرفت که در واقع هیچ کدام از این دو نسبت به دیگری حساس تر به نویز نیست و هر دو رفتار یکسانی از خود نشان می دهند.

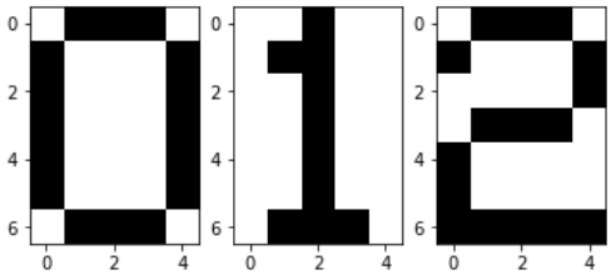
قسمت 4:

برای این قسمت نیز برای رسیدن به تصاویری که برخی از پیکسل های آن ها صفر است، از توابع سوال 1 استفاده شده است.

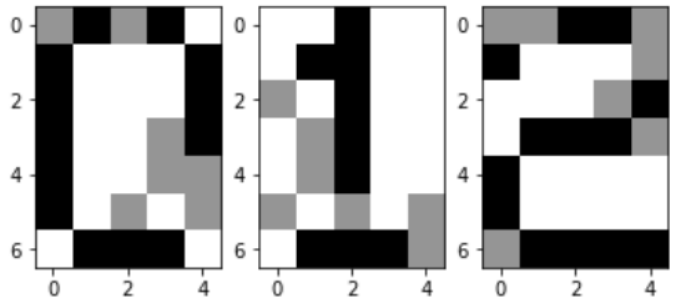
از دست رفتن 20 درصد مقادیر:



خروجی های شبکه

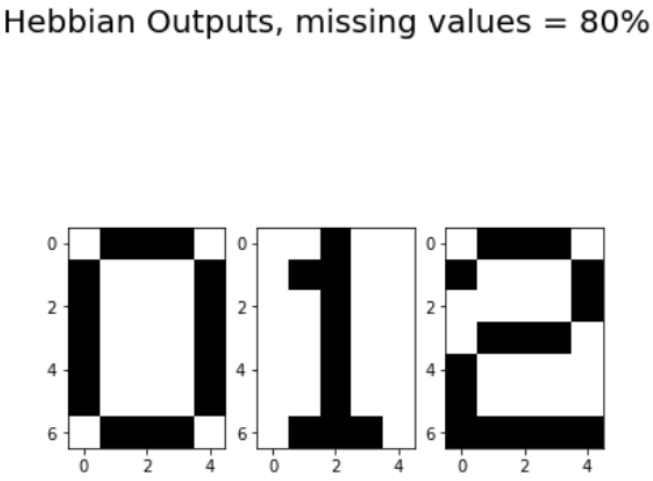


خروجی های مطلوب

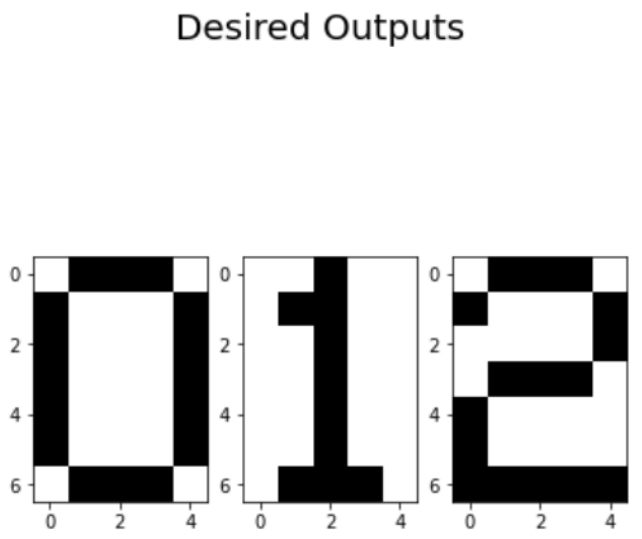


ورودی های با پیکس هایی از دست رفته

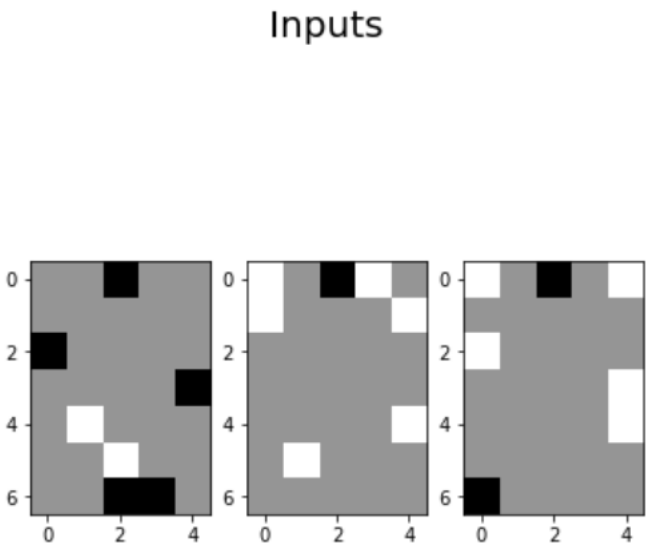
از دست رفتن 80 درصد مقادیر:



خروجی های شبکه



خروجی های مطلوب



ورودی های با پیکس هایی از دست رفته

با توجه به نتایج قسمت های 3 و 4 مشاهده می شود که شبکه طراحی شده در مقابل از دست رفتن پیکسل ها کاملا مقاوم است. در حالت اعمال نویز با نویز 80 درصد، شبکه پاسخ کاملا اشتباه داد اما وقتی پیکسل ها را به صورت تصادفی صفر می کنیم هم برای حالت 20 درصد و هم برای 80 درصد، خروجی های شبکه و خروجی های مطلوب کاملا یکسان بودند.

یادآوری: برای این سوال از توابع و کدهای سوال اول استفاده شده است. با ان تفاوت که برای به دست آوردن وزن ها دیگر به خروجی نیازی نبود و تابعی جدید برای به دست آوردن وزن های modified hebbian تعریف شد که در آن ماتریس همانی در هر مرحله از وزن ها کم می شد. در ادامه نیز برای اعمال نویز و از دست دادن مقادیر در عکس ها، از توابع سوال 1 استفاده شد. برای خود شبکه نیز ورودی outputرا از تابع آن حذف کردم.

سوال ۳ – Discrete Hopfield Network

قسمت 1:

در قسمت قبل الگوریتم Hebbian را برای Auto association اجرا کردیم و دیدیم که این الگوریتم برای زمانی که مقدار نویز بسیار زیاد است، کارایی خوبی ندارد. برای رفع این مشکل discrete hpfield network پیشنهاد شد. در این الگوریتم مجددا وزن ها از طریق modified hebbian به دست می آیند با این تفاوت که در هر مرحله یک یونین از ورودی آپدیت می شود و با وردی متناظرش جمع می شود. در ادامه مراحل الگوریتم به دقت توضیح داده شده است:

مرحله 0: مقدار دهی ماتریس وزن با استفاده از روش modified hebbian lerarning rule. تا وقتی که شرط همگرایی محقق نشده مراحل 1 تا 7 را انجام دهید.

مرحله 1: به ازای هر ورودی مراحل 2 تا 6 انجام شود.

مرحله 2: yi = x1

مرحله 3: مراحل 4 تا 6 را برای هر یونیت Yi انجام دهید.

مرحله 4: y_in1 = xi + sigma(yj, weightji)

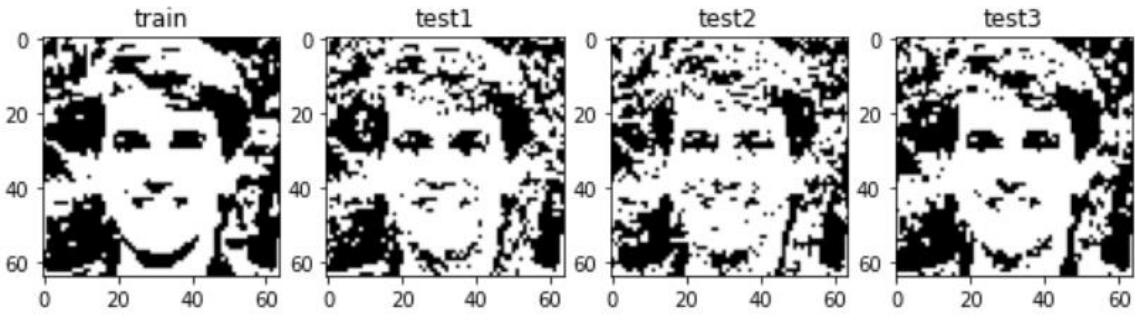
مرحله 5: yi = Activation(y_ini)

مرحله 6: ورودی activation را بر اساس خروجی بخش قبل آپدیت کنید

مرحله 7: چک کردن شرط همگرایی.

قسمت 2:

توضیح کد: در این بخش برای تغییر سایز و تبدیل کردن عکس ها به bipolar تابعی به نام normalize نوشته شده است. در این تابع ابتدا بعد سوم حذف می شود. مقدار تمام پیکسل ها به 255 تقسیم می شود و پس از امتحان کردن threshold های مختلف مقدار ایده ال و مطلوب برای threshold برابر 0.46 در نظر گرفته شد. لازم به یادآوری است که در این تابع مقادیر بزرگتر از 0.46 به 1- و مقادیر کمتر به 1 مقداردهی شده اند. برای اینکه تصویر سیاه و سفید حالت مطلوبی داشته باشد.



تصاویر حاصل بعد از عبور از تابع normalize

قسمت 3:

برای به دست آوردن مقادیر ماتریس وزن ها در این سوال نیز از تابع سوال قبل استفاده شده است.

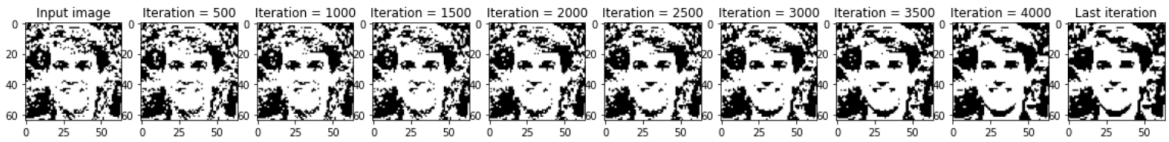
```
train_input = train.reshape(M, 1)
weight = MdHebbian_Weight(train_input)
print(weight)
```

```
[[ 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.]]
```

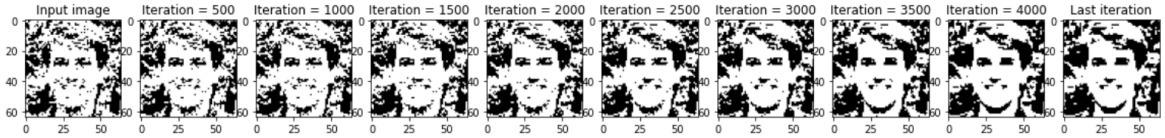
ماتریس وزن های شبکه

قسمت 4:

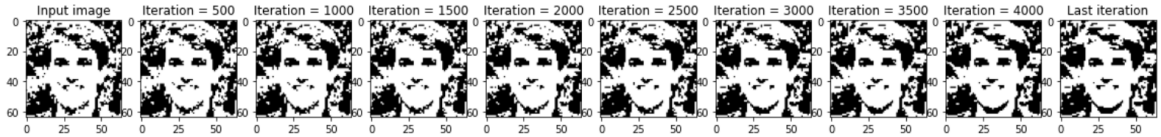
توضیح کد: در این مرحله برای اجرای الگوریتم یادگیری حلقه ای به طول 4096 تکرار می شود تا زمانی که شرط همگرایی برقرار شود. هر 50 تا یونیت که آپدیت می شود در لیستی تصویر نهایی ذخیره می شود. مقدار hamming نیز به ازای آپدیت شدن 50 تا یونیت ذخیره می شود. در نهایت اگر شرط همگرایی با یک بار آپدیت کرن یونیت ها محقق شده بود در لیست نهایی، 81 تصویر خواهیم داشت. تصاویر رسم شده، تصاویری با اندیس های 10و 20و 30و ...و 80 و آخرین تصویر آپدیت شده است.



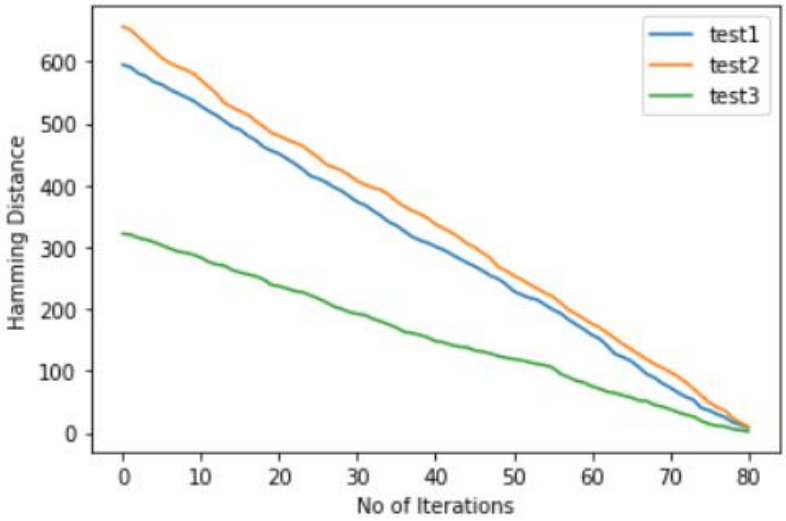
تصاویر خروجی شبکه برای test1



تصاویر خروجی شبکه برای test2



تصاویر خروجی شبکه برای test3



نمودار hamming distance برای هر سه تصویر تست

سوال ۴ – Bidirectional Associative Memory

قسمت 1:

توضیح کد:

در این بخش ابتدا کد اسکی تمام کلمات را به صورت آرایه ذخیره کردم. به جای درایه های 0، عدد -1 را قرار دادم که تابع فعالسازی آن مانند بخش های قبل باشد. در ادامه ماتریس وزن ها نیز با روش بخش های قل و از حاصل جمع ضرب ورودی ها و خروجی ها به دست آمد.

```
W = np.dot(X1, Y1) + np.dot(X2, Y2) + np.dot(X3, Y3)
W
```

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

ماتریس وزن به دست آمده

قسمت 2:

توضیح کد: در این بخش برای هر زوج X, Y یک بار X ره به عنوان ورودی به شبکه دادم و خروجی به دست آمده را با Y متناظر مقایسه کردم. در مرحله دوم نیز برعکس این کار را انجام دادم یعنی Y را به عنوان ورودی شبکه دادم و خروجی به دست آمده را با X متناظر آن مقایسه کردم.

نتایج:

```
#Clinton-President
t_in = np.dot(np.transpose(X1), W)
output = np.zeros((1, 63))
output = np.where(t_in > 0, 1, -1)
if(np.array_equal(output, Y1)):
  print("Works correctly for Clinton-President pair (Y detecting)")
```

```
Works correctly for Clinton-President pair (Y detecting)
```

نتیجه تشخیص Y برای زوج اول

```
#Kenstar-Gentelman
t_in = np.dot(np.transpose(X2), W)
output = np.zeros((1, 63))
output = np.where(t_in > 0, 1, -1)
if(np.array_equal(output, Y2)):
  print("Works correctly for Kenstar-Gentelman pair (Y Detecting)")
```

```
Works correctly for Kenstar-Gentelman pair (Y Detecting)
```

نتیجه تشخیص X برای زوج اول

```
#Hillay-FirstLady
t_in = np.dot(np.transpose(X2), W)
output = np.zeros((1, 63))
output = np.where(t_in > 0, 1, -1)
if(np.array_equal(output, Y2)):
  print("Works correctly for Hillay-FirstLady pair (Y Detecting)")
```

```
Works correctly for Hillay-FirstLady pair (Y Detecting)
```

نتیجه تشخیص Y برای زوج دوم

```
#Hillay-FirstLady
t_in = np.dot(Y2, W.T)
output = np.zeros((1, 49))
output = np.where(t_in > 0, 1, -1)
if(np.array_equal(output, np.transpose(X2))):
  print("Works correctly for Hillay-FirstLady pair (X detecting)")
```

```
Works correctly for Hillay-FirstLady pair (X detecting)
```

نتیجه تشخیص X برای زوج دوم

نتیجه تشخیص Y برای زوج سوم

```
#Kenstar-Gentelman
t_in = np.dot(Y3, W.T)
output = np.zeros((1, 49))
output = np.where(t_in > 0, 1, -1)
if(np.array_equal(output, np.transpose(X3))):
  print("Works correctly for Kenstar-Gentelman pair (X detecting)")
```

```
Works correctly for Kenstar-Gentelman pair (X detecting)
```

نتیجه تشخیص X برای زوج سوم

قسمت 3:

توضیح کد: در این بخش یک حلقه 100 تایی نوشتم. در این حلقه هر بار ورودی شبکه نویزی می شود با درصد مورد نظر. سپس این ورودینویزی به شبکه داده می شود تا الگوریتم روی آن اجرا شود. این الگوریتم تا زمانی ادامه پیدا می کند که یا خروجی شبکه به خروجی اصلی همگرا شود و یا اینکه 100 بار الگوریتم تکرار شود. این کار هم در جهت رفت و هم در جهت برگشت انجام شد. محاسبه درصد شباهت نیز به این صورت انجام شد که هربار که از الگوریتم خارج می شد، تعداد بیت های برابر خروجی شبکه با خروجی اصلی مقایسه می شد و تقسیم بر تعداد بیت های خروجی و در 100 ضرب می شد. در نهایت یک لیست 100 تایی داریم که هرکدام درصد شباهت خروجی شبکه و خروجی اصلی را برای هربار اجرای الگوریتم دارند که روی این مقادیر میانگین گرفته می شود.

جدول نتایج:

جدول درصد تشابه خروجی شبکه و خروجی اصلی با اعمال 10 درصد نویز

جهت اعمال نویز	زوج اول	زوج دوم	زوج سوم
رفت	100	100	100
برگشت	100	100	100

جدول درصد تشابه خروجی شبکه و خروجی اصلی با اعمال 20 درصد نویز

جهت اعمال نویز	زوج اول	زوج دوم	زوج سوم
رفت	98.5	99.84	99.42
برگشت	99.67	99.32	99.83

قسمت 4:

توضیح کد: در این بخش مجددا مانند قسمت اول کدهای اسکی ورودی جدید را ذخیره کردم و ماتریس وزن جدیدی تولید کردم. نتایج به صورت زیر است:

```
noOfcorrect = 0
for i in range(N):
  y_in = np.dot(np.transpose(X[i]), new_W)
  out = np.zeros((1, 63))
  out = np.where(y_in > 0, 1, -1)
  if(np.array_equal(out, Y[i])):
    noOfcorrect  = noOfcorrect + 1
  else:
    print(f'New network can not detect output correctly for {pairs[i]} when X is input')

print(f'New network can detect {noOfcorrect} outputs correctly when X is input')
```

```
New network can detect 4 outputs correctly when X is input
```

امتحان کردن شبکه جدید با دادن X به عنوان ورودی

```
noOfcorrect = 0
for i in range(N):
  y_in = np.dot(Y[i], new_W.T)
  out = np.zeros((1, 49))
  out = np.where(y_in > 0, 1, -1)
  if(np.array_equal(out, np.transpose(X[i]))):
    noOfcorrect  = noOfcorrect + 1
  else:
    print(f'New network can not detect output correctly for {pairs[i]} when Y is input')

print(f'New network can detect {noOfcorrect} outputs correctly when Y is input')
```

```
New network can not detect output correctly for Lewisky-SweetGirl when Y is input
New network can detect 3 outputs correctly when Y is input
```

امتحان کردن شبکه جدید با دادن Y به عنوان ورودی