به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی و یادگیری عمیق

تمرین سری سوم

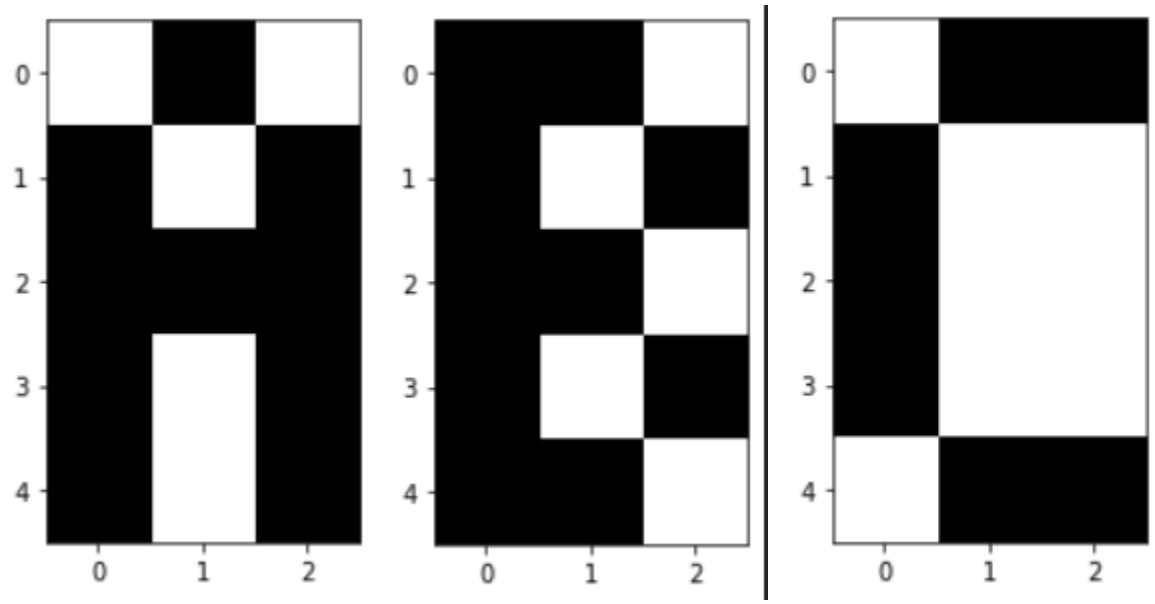
نام و نام خانوادگی	علی ایزدی
شماره دانشجویی	۸۱۰۱۹۹۱۰۲
تاریخ ارسال گزارش	۳ دی ۱۴۰۰

سوال ۱- Character Recognition using Hebbian Learning Rule

الف)

شبکه مطابق با hebbian Rule آموزش داده شده است که وزن های شبکه از جمع ضرب خارجی خروجی و ورودی های داده شده به دست می آید.

در ادامه ورودی های داده شده به شبکه داده شده اند و خروجی شبکه به ازای حروف مختلف در زیر آورده شده است.



همان طور که مشاهده میشود شبکه توانسته است خروجی های مطلوب را با ورودی های داده شده تولید کند.

ب)

چون ۳ حرف مختلف داریم پس میتوان یک ماتریس ۲ در ۱ در نظر گرفت و خروجی ها را مطابق با زیر تعریف کرد:

```
At_s = np.array([[1],[1]])
Bt_s = np.array([[-1],[1]])
Ct_s = np.array([[1],[-1]])
```

سپس مطابق با قبل اگر شبکه آموزش داده شود و خروجی ها به ازای ورودی های داده شده ایجاد شوند مشاهده میشود که شبکه توانسته است به خروجی های تعریف شده در بالا برسد:

```
    smallest_targets = [At_s, Bt_s, Ct_s]

    hebbian = Hebbian()
    hebbian.fit(inputs, smallest_targets)
    outputs = hebbian.transform(inputs)

    for output in outputs:
        print(output)
```

✓ 0.4s

```
[[1 1]]
[[-1  1]]
[[ 1 -1]]
```

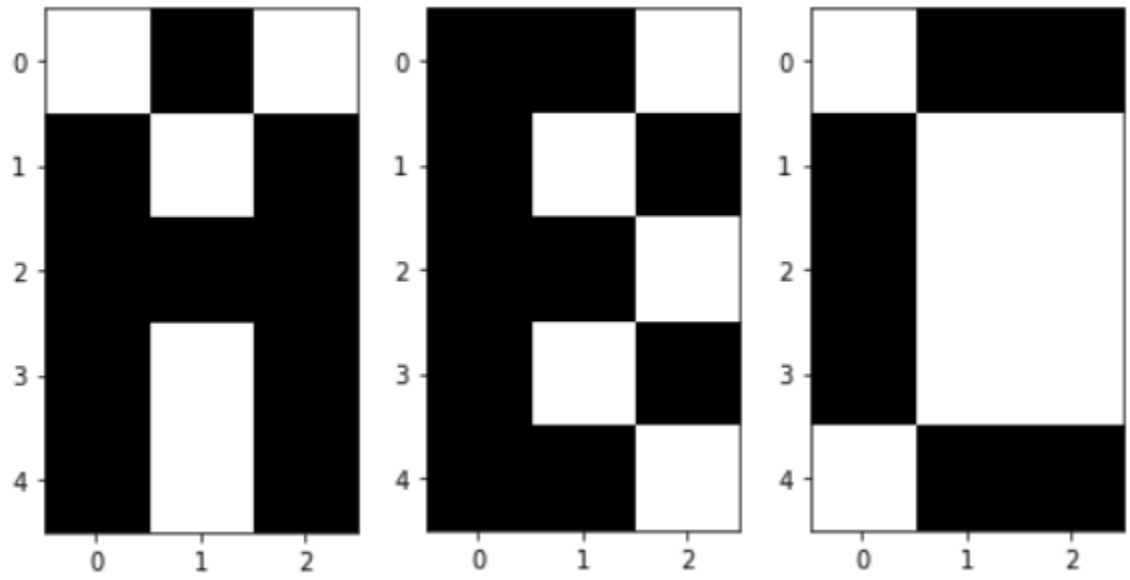
پ)

برای اضافه کردن نویز با درصد مشخص به این صورت عمل میکنیم که روی همه عناصر iterate میکنیم و برای هر عنصر یک عدد تصادفی بین ۰و ۱ تولید میکنیم که اگر مقدار آن کمتر از مقدار نویز مشخص شده بود آن عنصر را از ۱ به ۱- یا بر عکس تبدیل مکنیم و در غیر این صورت عنصر بدون تغییر میماند.

```
for i in range(len(s)):
    for j in range(len(s[i])):
        if np.random.rand() < self.noise:
            s[i][j] = -s[i][j]
```

در زیر یک بار ورودی نویز داده شده به ازای درصد نویز ۱۰ و ۴۰ درصد و خروجی های ۵ در ۳ و هم چنین ۲ در ۱ قسمت ب رسم شده اند.

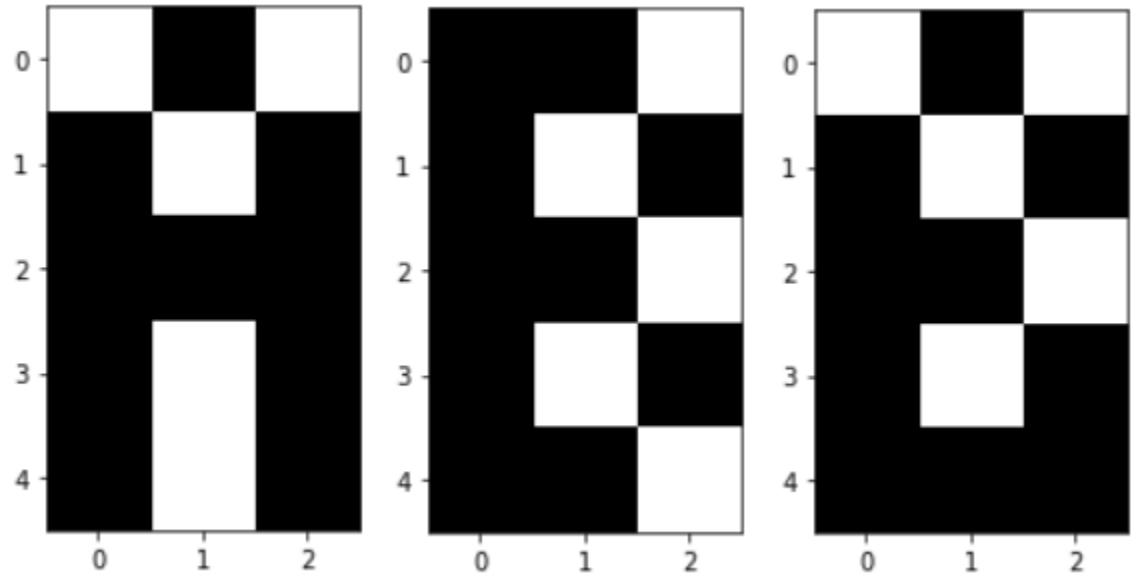
حالت اول) نویز ۱۰ درصد و خروجی ۵ در ۳:
خروجی ها به صورت صحیح تولید شده اند.



حالت دوم) نویز ۱۰ درصد و خروجی ۲ در ۱:
خروجی ها به صورت صحیح تولید شده اند.

```
[[1 1]]
[[-1  1]]
[[ 1 -1]]
```

حالت سوم) نویز ۴۰ درصد و خروجی ۵ در ۳:
همان طور که مشاهده میشود حرف A به صورت صحیح و Bو C غلط ایجاد شده اند.



حالت چهارم) نویز ۴۰ درصد و خروجی ۲ در ۱:
در این حالت حرف A اشتباه و حرف B, C درست ایجاد شده اند (گرچه به ازای اجراهای مختلف نتایج مختلفی گرفته میشود که در ادامه بررسی خواهیم کرد.)

```
[[1 1]]
[[-1  1]]
[[ 1 -1]]
```

در ادامه حالت های بالا را به ازای ۱۰۰۰ اجرای مختلف بررسی میکنیم که در چند درصد مواقع خروجی درست تشخیص داده میشود.

حالت اول) نویز ۱۰ درصد و خروجی ۵ در ۳:
تقریبا در اکثر مواقع سه حرف درست تشخیص داده میشوند.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 99.9% times
--------------------
C-true detected in : 99.6% times
--------------------
```

حالت دوم) نویز ۱۰ درصد و خروجی ۲ در ۱:
مشابه حالت قبل تقریبا در همه مواقع درست تشخیص داده میشوند.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 99.8% times
--------------------
C-true detected in : 99.9% times
--------------------
```

حالت سوم) نویز ۴۰ درصد و خروجی ۵ در ۳:
تقریبا در ۵۰ درصد مواقع درست تشخیص داده میشوند.

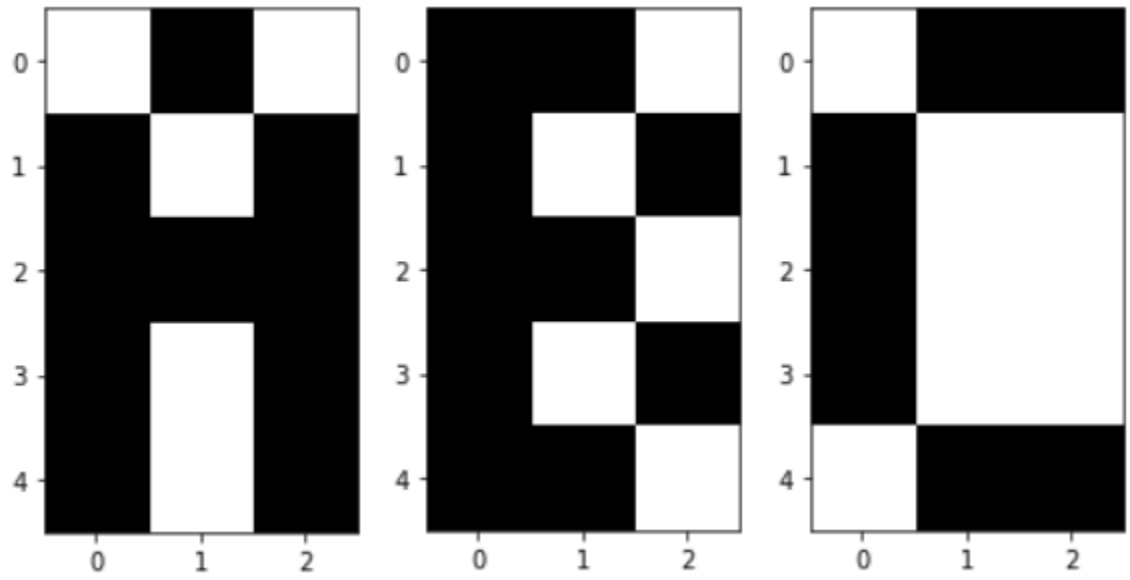
```
A-true detected in : 52.7% times
--------------------
B-true detected in : 43.5% times
--------------------
C-true detected in : 45.9% times
--------------------
```

حالت چهارم) نویز ۴۰ درصد و خروجی ۲ در ۱:
نسبت به حالت خروجی ۵ در ۳ در این حالت درصد مواقع تشخیص درست افزایش یافته است.

```
A-true detected in : 76.3% times
--------------------
B-true detected in : 54.1% times
--------------------
C-true detected in : 54.1% times
--------------------
```

ت)
برای از بین بردن اطلاعات با درصد مشخص به این صورت عمل میکنیم که روی همه عناصر iterate میکنیم و برای هر عنصر یک عدد تصادفی بین ۰و ۱ تولید میکنیم که اگر مقدار آن کمتر از مقدار درصد مشخص شده بود آن عنصر را از ۱و ۱- به ۰ تبدیل میکنیم و در غیر این صورت عنصر بدون تغییر میماند.

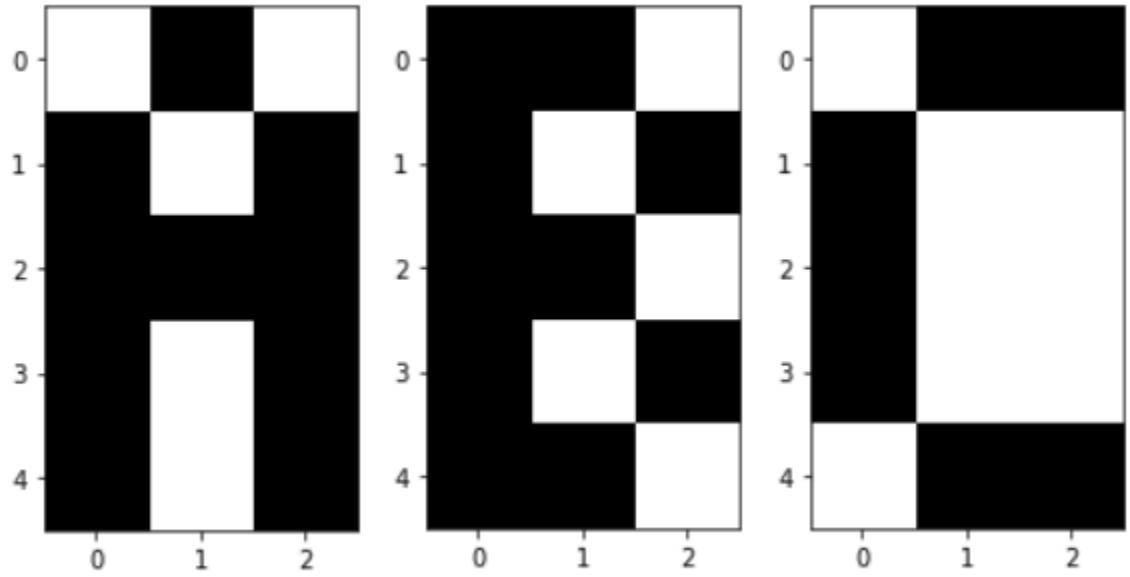
حالت اول) از بین رفتن ۱۰ درصد و خروجی ۵ در ۳:
خروجی ها درست تشخیص داده شده اند.



حالت دوم) از بین رفتن ۱۰ درصد و خروجی ۲ در ۱:
خروجی ها درست تشخیص داده شده اند.

```
[[1 1]]
[[-1  1]]
[[ 1 -1]]
```

حالت سوم) از بین رفت ۴۰ درصد و خروجی ۵ در ۳:
خروجی ها درست تشخیص داده شده اند.



حالت چهارم) از بین رفتن ۴۰ درصد و خروجی ۲ در ۱:
خروجی ها درست تشخیص داده شده اند.

```
[[1 1]]
[[-1  1]]
[[ 1 -1]]
```

در ادامه حالت های بالا را به ازای ۱۰۰۰ اجرای مختلف بررسی میکنیم که در چند درصد مواقع خروجی درست تشخیص داده میشود.

حالت اول) از بین رفتن ۱۰ درصد و خروجی ۵ در ۳:
در همه مواقع درست تشخیص داده شده است.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 100.0% times
--------------------
C-true detected in : 100.0% times
--------------------
```

حالت دوم) از بین رفتن ۱۰ درصد و خروجی ۲ در ۱:
در همه مواقع درست تشخیص داده شده است.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 100.0% times
--------------------
C-true detected in : 100.0% times
--------------------
```

حالت سوم) از بین رفت ۴۰ درصد و خروجی ۵ در ۳:
در اکثر مواقع درست تشخیص داده شده است.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 99.9% times
--------------------
C-true detected in : 99.9% times
--------------------
```

حالت چهارم) از بین رفتن ۴۰ درصد و خروجی ۲ در ۱:
در اکثر مواقع درست تشخیص داده شده است.

```
A-true detected in : 100.0% times
--------------------
B-true detected in : 99.9% times
--------------------
C-true detected in : 99.9% times
--------------------
```

د)
مطابق با نتایج مرحله قبل مقاومت شبکه در برابر اضافه کردن نویز بیشتر از از بین بردن اطلاعات است.

هم چنین در حالت اضافه کردن نویز کاهش اندازه خروجی باعث افزایش ۱۰ تا ۳۰ درصدی دقت خروجی شده است.
در حالت از بین بردن اطلاعات نیز دقت با کاهش خروجی افزایش یافته است. (البته چون شبکه در برابر از بین بردن اطلاعات مقاومت داشته است در هر دو حالت خروجی با اندازه های مختلف دقت شبکه بسیار بالا و نزدیک به ۱۰۰ درصد است.)

سوال ۲- Auto-associative Net

۱)

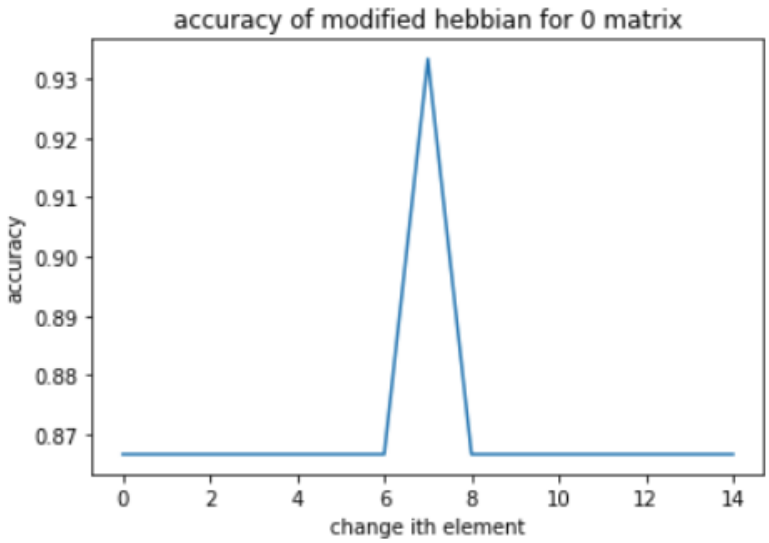
شبکه مطابق با modified hebbian rule آموزش داده شده است که وزن های شبکه از جمع ضرب خارجی خروجی و ورودی های داده شده منهای ماتریس قطری با قطر اندازه تعداد داده ها که در این جا ۲ است مطابق زیر به دست می آید.

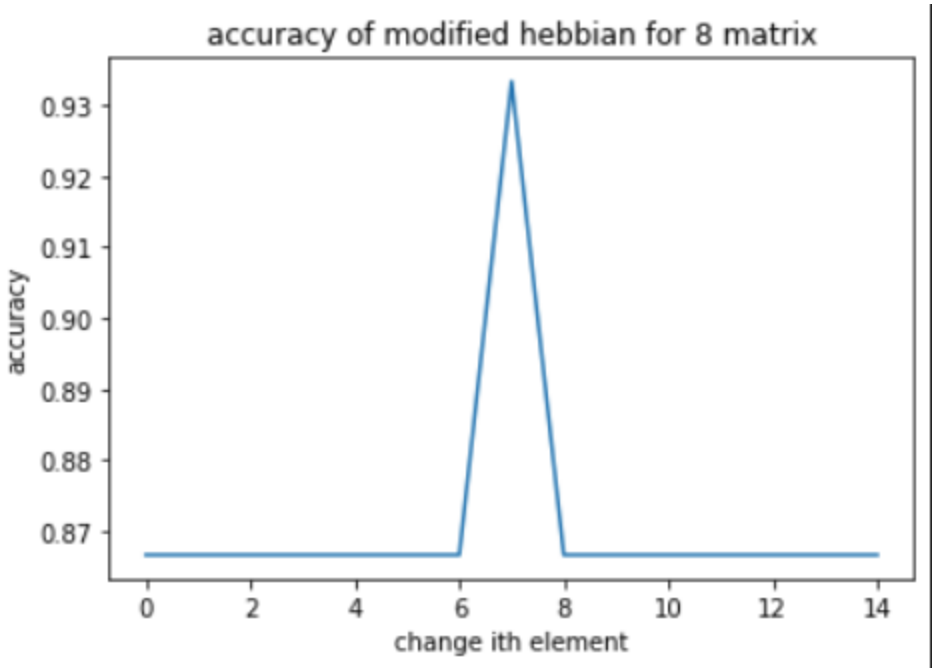
```
self.W = np.sum([s @ t.T for s, t in zip(inputs, inputs)], axis=0)
self.W = self.W - len(inputs) * np.identity(self.W.shape[0])
```

۲)

ابتدا درصد قوام شبکه در برابر یک اشتباه در ورودی را بررسی میکنیم.

برای این کار ۱۵ عنصر ماتریس داده شده را به از بالا به پایین و از چپ به راست به صورت سطری یکی یکی تغییر میدهیم و دقت شبکه را برای این ۱۵ حالت برای عدد ۸ و ۰ رسم میکنیم. دقت در این جا به معنای تعداد خانه های درست تشخیص داده شده از میان ۱۵ خانه است.



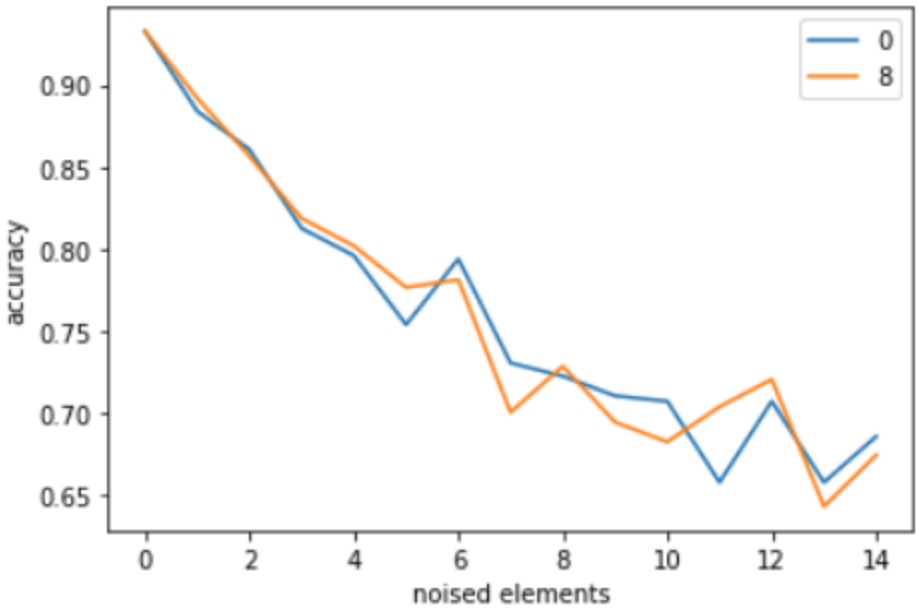


همان طور که مشاهده میشود. برای هر دو ماتریس ۰ و ۸ تغییر یک خانه منجر به دقت ۰.۸۷ در بازیابی ماتریس اصلی شده است و تنها برای یک حالت تغییر عنصر یعنی عنصر ۸ام که عنصر وسط ماتریس است دقت افزایش یافته است.

نتایج بالا تنها برای تغییر یکی از عناصر ماتریس و بررسی همه حالت های تغییر یک عنصر یعنی ۱۵تا بود.

در ادامه اما میخواهیم حالات مختلف تعداد اشتباه در ورودی از ۱ تا ۱۵ را بررسی کنیم.

برای این کار برای حالت های مختلف p=1…15 از n=15 از توزیع چند جمله ای با احتمال موفقیت نیم تعداد ۱۰۰ نمونه تولید میکنیم که در واقع ۱۰۰ تا نمونه به ازای هر یک از حالت های p تولید میکنیم که اگر مثلا p=۳ باشد ۱۰۰ نمونه ۳ تایی تولید کرده ایم.
سپس دقت را برای این ۱۰۰ نمونه نویز داده شده حساب میکنیم و میانگین میگیریم.
در نتیجه نمودار زیر میانگین دقت برای ۱ تا ۱۵تا عنصر نویز داده شده را نشان میدهد.

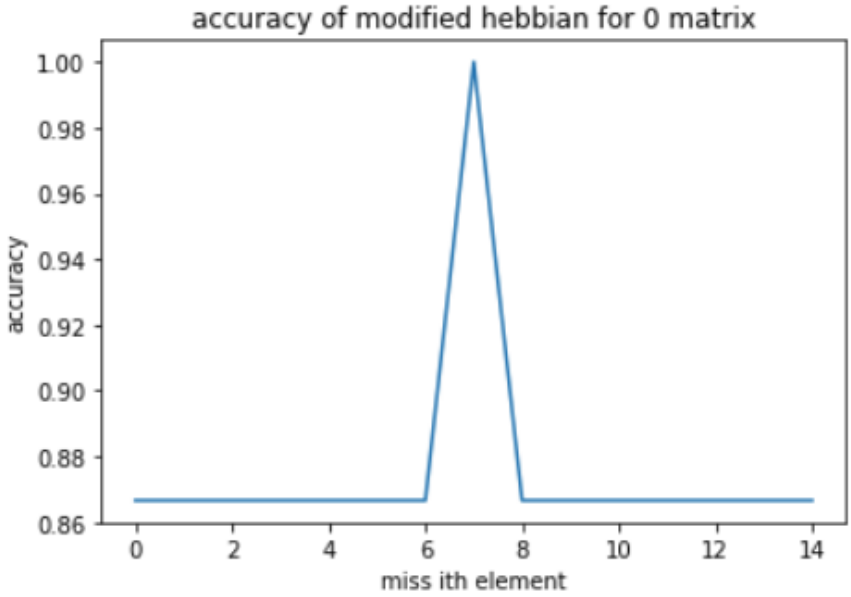


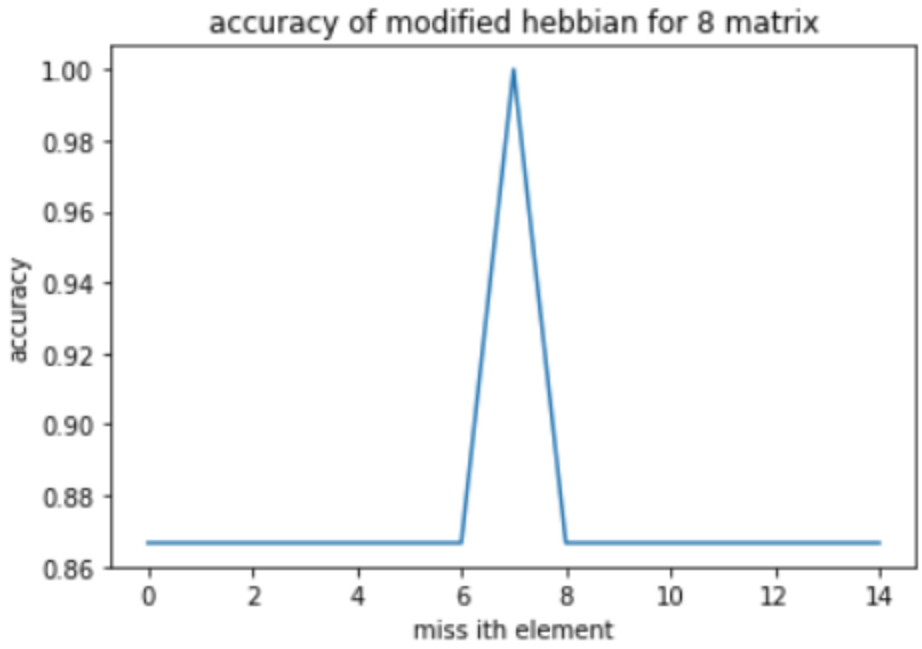
همان طور که در بالا مشاهده میشود با افزایش تعداد عناصر نویز داده شده دقت برای هر دو عدد ۰و ۸ به صورت تقریبا یکسان کاهش میابد.

۳)

درصد قوام شبکه در برابر از بین رفتن یک عنصر در ورودی را بررسی میکنیم.

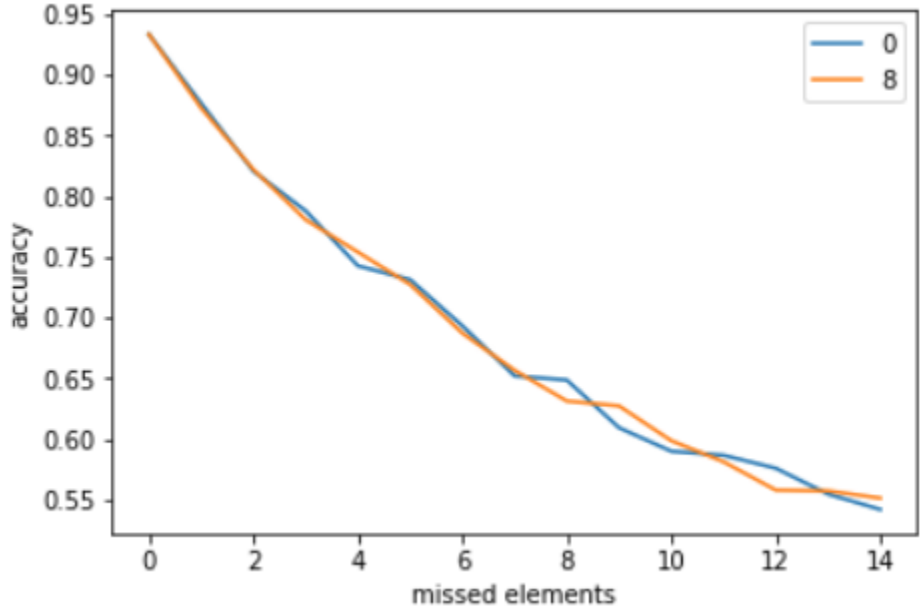
برای این کار ۱۵ عنصر ماتریس داده شده را به از بالا به پایین و از چپ به راست به صورت سطری یکی یکی صفر میکنیم و دقت شبکه را برای این ۱۵ حالت برای عدد ۸ و ۰ رسم میکنیم. دقت در این جا به معنای تعداد خانه های درست تشخیص داده شده از میان ۱۵ خانه است.





همان طور که مشاهده میشود. برای هر دو ماتریس ۰ و ۸ صفر کردن یک خانه منجر به دقت ۰.۸۷ در بازیابی ماتریس اصلی شده است و تنها برای یک حالت تغییر عنصر یعنی عنصر ۸ام که عنصر وسط ماتریس است دقت افزایش یافته و به ۱۰۰ رسیده است.

در ادامه اما میخواهیم حالات مختلف تعداد از بین بردن اطلاعات عناصر در ورودی از ۱ تا ۱۵ را بررسی کنیم.



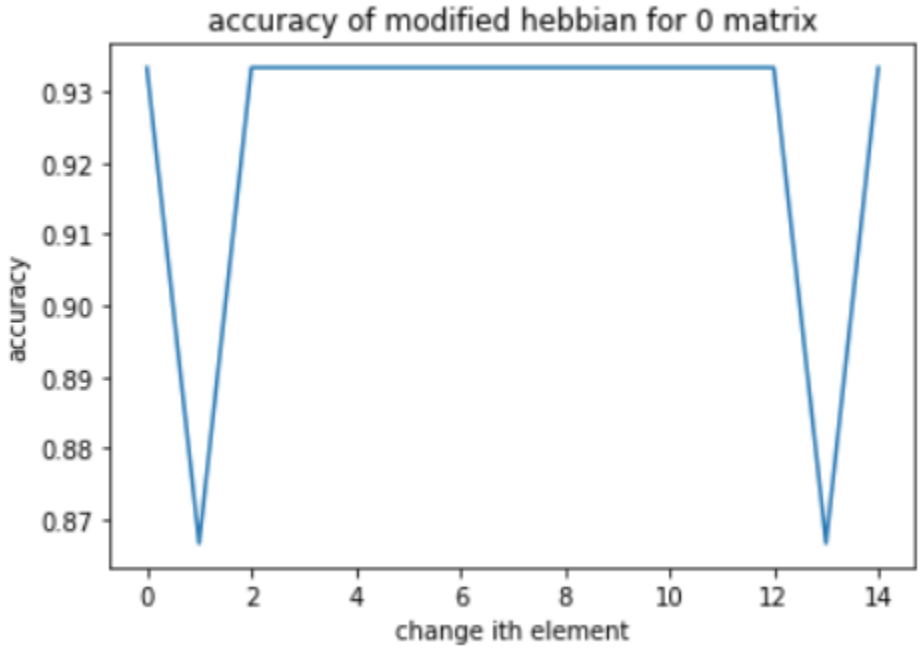
همان طور که در بالا مشاهده میشود با افزایش تعداد عناصر از بین رفته اطلاعات دقت برای هر دو عدد ۰و ۸ به صورت تقریبا یکسان کاهش میابد.

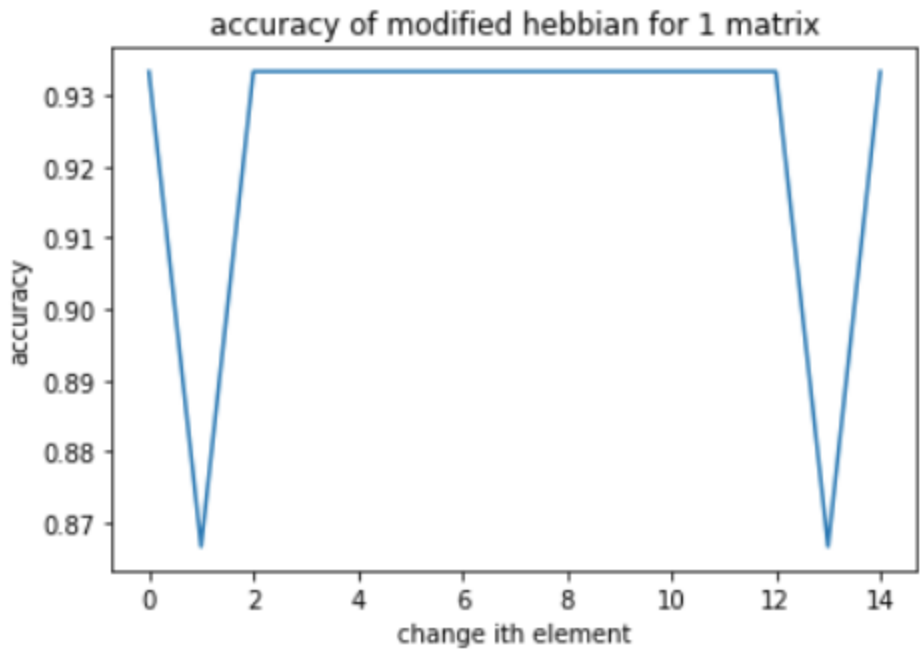
دقت زمانی که اطلاعت زیادی از بین رفته است کمتر است نسبت به حالتی که اطلاعات زیادی نویز داده شده اند.

نمیتوان نتیجه را تعمیم داد و وابسته به پترن ورودی و داده ها میتواند باشد.

۴)

ابتدا درصد قوام شبکه در برابر یک اشتباه در ورودی را بررسی میکنیم.
برای این کار ۱۵ عنصر ماتریس داده شده را به از بالا به پایین و از چپ به راست به صورت سطری یکی یکی تغییر میدهیم و دقت شبکه را برای این ۱۵ حالت برای عدد 1 و ۰ رسم میکنیم. دقت در این جا به معنای تعداد خانه های درست تشخیص داده شده از میان ۱۵ خانه است.

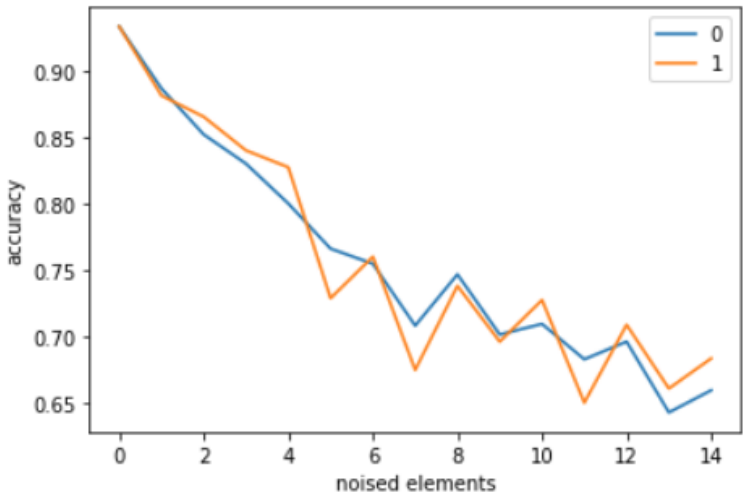




همان طور که مشاهده میشود. برای هر دو ماتریس ۰ و ۱ تغییر یک خانه منجر به دقت ۰.۹۳ در بازیابی ماتریس اصلی شده است و تنها برای دو حالت تغییر عنصر یعنی عنصر دوم و ۱۴ام دقت کاهش یافته است.

نتایج بالا تنها برای تغییر یکی از عناصر ماتریس و بررسی همه حالت های تغییر یک عنصر یعنی ۱۵تا بود.

در ادامه اما میخواهیم حالات مختلف تعداد اشتباه در ورودی از ۱ تا ۱۵ را بررسی کنیم.



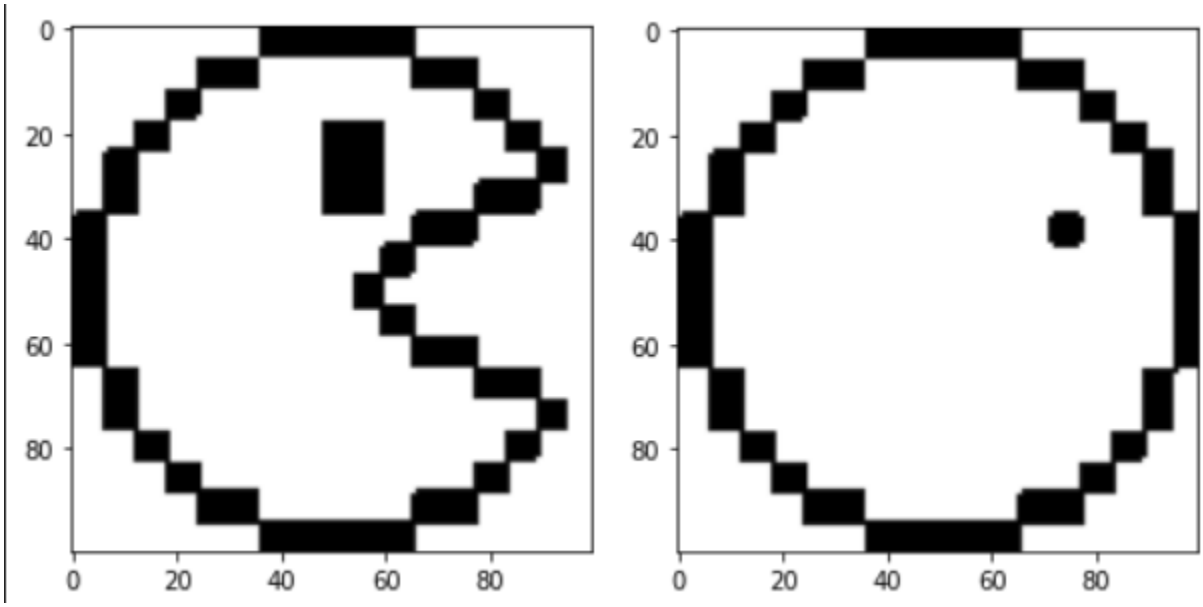
همان طور که در بالا مشاهده میشود با افزایش تعداد عناصر نویز داده شده دقت برای هر دو عدد ۰و ۱ به صورت تقریبا یکسان کاهش میابد.

برای تعداد عناصر نویز داده شده کمتر در این حالت دقت نسبت به دو ماتریس ۰و ۸ بیشتر است زیرا در اینجا دو پترن ۰و ۱ راحت تر قابل تشخیص از هم هستند تا دو ماتریس ۰و ۸

سوال ۳- Discrete Hopfield Net

۱)

با اجرای کد داده شده نتایج زیر نمایش داده میشوند.



۲)

شبکه مطابق با hebbian Rule استفاده از عکس دهان باز آموزش داده شده است که وزن های شبکه از جمع ضرب خارجی خروجی و ورودی های داده شده به دست می آید. هم چنین قطر اصلی طبق روش BAM باید صفر شود. هم چنین چون ورودی binary است ضرب خارجی مطابق با زیر انجام میشود.(2x-1)

```
self.x = x
self.W = (2 * x - 1) @ (2 * x - 1).T
self.W = self.W - np.diag(np.diag(self.W))
```

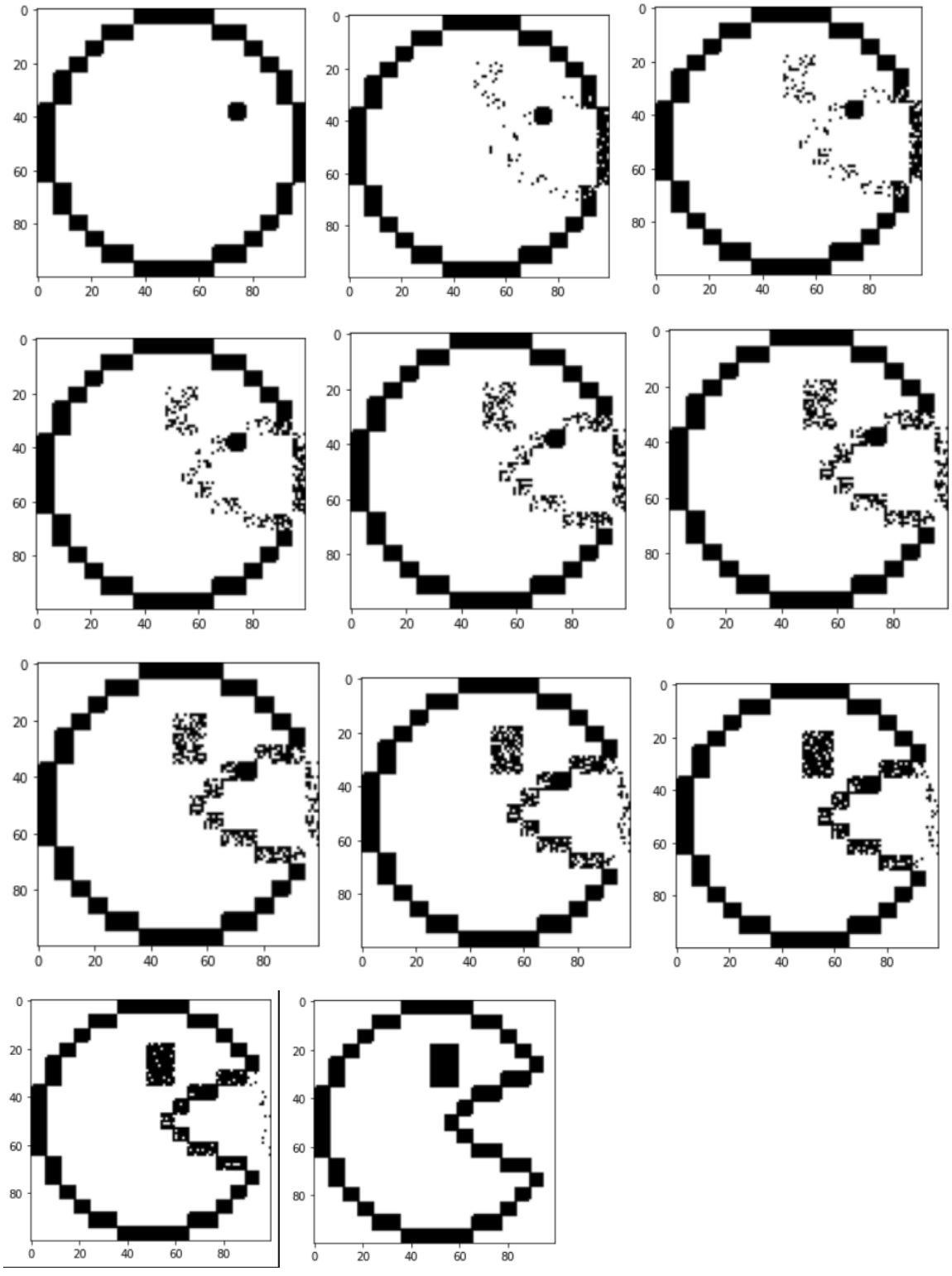
۳)

در این قسمت هدف دادن ورودی عکس دهان بسته به شبکه است تا در چندین iteration به عکس دهان باز برسد.

برای این کار مطابق با زیر در هر iteration به صورت رندوم یکی از عناصر عکس دهن بسته انتخاب شده است و مطابق با نحوه به روز رسانی الگوریتم و تابع فعال سازی آن عنصر به روز رسانی شده است.

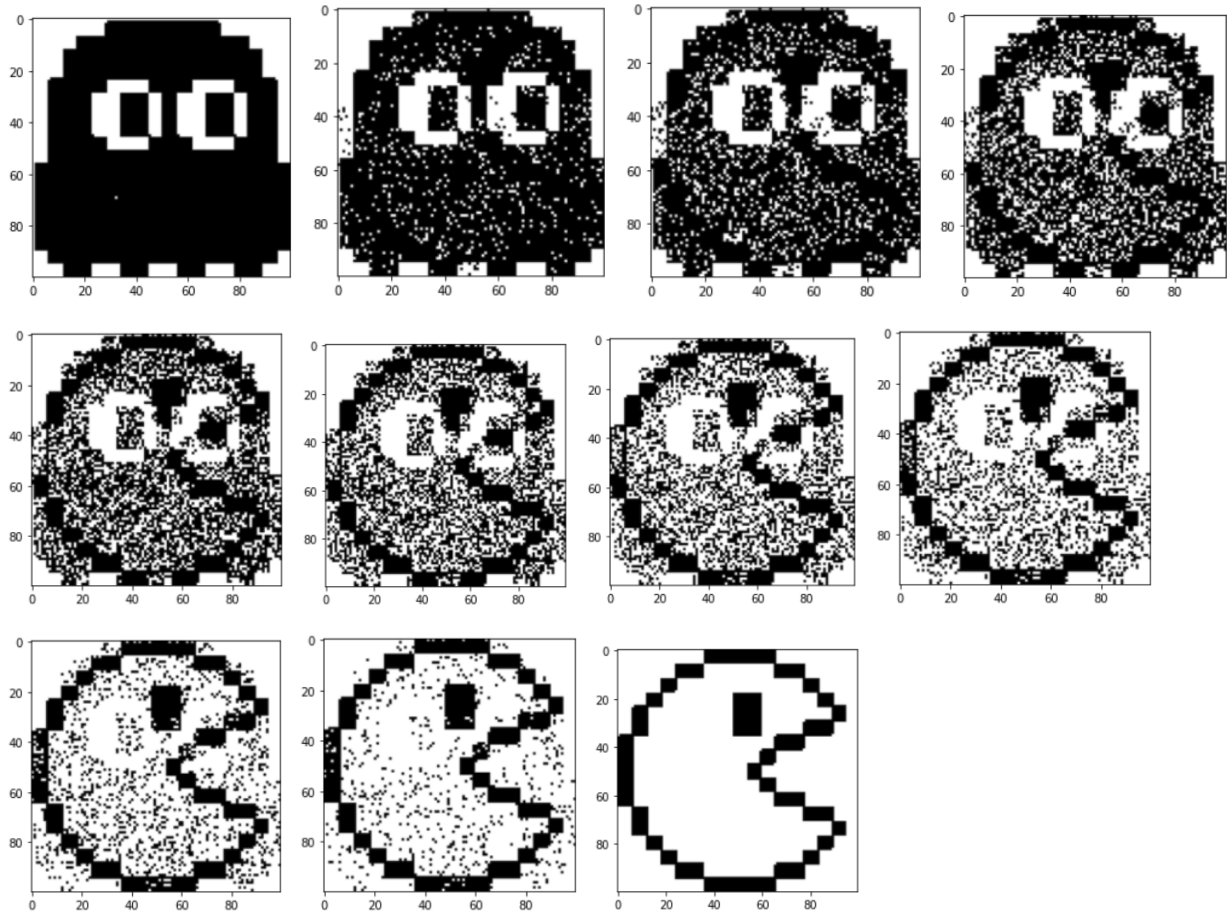
```
def active(y_in, out_i, threshold=0):
    if y_in > threshold:
        return 1
    elif y_in == threshold:
        return out_i
    elif y_in < threshold:
        return 0

accs = []

for it in range(n_iter):
    print('iteration: ', it)
    idx = np.random.permutation(out.shape[0])
    for i, index in enumerate(idx):
        y_in = y[index] + out.T @ self.W[:, index]
        out[index] = active(y_in, out[index])
```

بعد از هر ۱۰۰۰ iteration نیز عکس خروجی در زیر آورده شده است که مشاهده میشود تقریبا بعد از ۱۱۰۰۰ مرحله به عکس دهان باز رسیده ایم.


0
20
40
60
80

۴)

برای عکس داده شده نیز transformation با استفاده از همان ماتریس وزن یادگرفته در مرحله قبل انجام شده است و بعد از هر ۱۰۰۰ iteration خروجی رسم شده است. مشاهده میشود تقریبا بعد از ۱۱۰۰۰ مرحله به عکس دهان باز رسیده ایم



سوال ۴- Bidirectional Associative Memory

الف)

شبکه مطابق با hebbian Rule آموزش داده شده است که وزن های شبکه از جمع ضرب خارجی خروجی و ورودی های داده شده به دست می آید.

در اینجا ورودی ها ماتریس های حروف داده شده و خروجی ها پترن های سه تایی با اعداد ۱ و ۱- است.

```
self.W = np.sum([s @ t.T for s, t in zip(inputs, targets)], axis=0)
```

ب)

برای تست هدف این است که میتوان از هر دوی ورودی و خروجی به یکدیگر رسید یا خیر.

Iterationهای الگوریتم مطابق با زیر پیاده سازی شده است که در هر مرحله x به y و y به x با تابع فعال ساز تبدیل شده است.

```
for it in range(n_iter):
    y_in = x_out.T @ self.W


    y_out = np.array([active(y_in.T[i], y_out[i]) for i in range(len(y_in.T))])
    y_out = y_out.reshape(y_out.shape[0], -1)

    x_in = y_out.T @ self.W.T
    x_out = np.array([active(x_in.T[i], x_out[i]) for i in range(len(x_in.T))])
    x_out = x_out.reshape(x_out.shape[0], -1)
```

تابع فعال ساز نیز مطابق با زیر پیاده سازی شده است.

```
def active(in_i, out_i, threshold=0):
    if in_i > threshold:
        return 1
    elif in_i == threshold:
        return out_i
    elif in_i < threshold:
        return -1
```

دقت ورودی و خروجی یعنی نسبت تعداد عناصر درست تشخیص داده شده برای هر ماتریس به کل برای هر کاراکتر در زیر آورده شده است.

```
C-input: 1.0
C-output: 1.0
E-input: 0.8
E-output: 0.6666666666666666
R-input: 1.0
R-output: 1.0
```

همان طور که مشاهده میشود دو حرف C و R پترن ورودی و خروجی آن درست بازیابی شده اند اما برای حرف E شبکه توانسته است ۸۰ درصد ورودی و ۶۶ درصد خروجی را بازیابی کند.

پ)

برای اضافه کردن نویز ۴۰ درصد به این صورت عمل میکنیم که روی همه عناصر iterate میکنیم و برای هر عنصر یک عدد تصادفی بین ۰و ۱ تولید میکنیم که اگر مقدار آن کمتر از ۰.۴ شده بود آن عنصر را از ۱ به ۱- یا بر عکس تبدیل میکنیم و در غیر این صورت عنصر بدون تغییر میماند.

نویز را به ورودی یعنی ماتریس ۵ در ۳ اعمال میکنیم و بررسی میکنیم خروجی ماتریس ۳ در ۱ در چند درصد مواقع دچار خطا میشود و نمیتواند ماتریس اصلی را بازیابی کند.
همان طور که در زیر مشاهده میشود حرف C بیشتر از همه و در ۶۴ درصد مواقع خروجی صحیح داده است سپس حرف R در ۳۱ درصد مواقع و حرف E هیچ گاه نتوانسته است با ۴۰ درصد نویز در ورودی خروجی صحیح را ایجاد کند.

```
C-output: 0.64
--------------------
E-output: 0.0
--------------------
R-output: 0.31
--------------------
```

ت)
چون خروجی ۸ حالت مختلف میتواند بگیرد پس تنها ۸ پترن در مجموع میتوان ذخیره کرد.

ث)
حال حروف O و F و P را نیز به شبکه اضافه میکنیم دوباره آن را آموزش میدهیم و مشابه با قسمت ب توانایی شبکه در بازیابی ورودی و خروجی را بررسی میکنیم.

```
C-input: 1.0
C-output: 1.0
E-input: 1.0
E-output: 0.6666666666666666
R-input: 0.7333333333333333
R-output: 0.6666666666666666
O-input: 0.8666666666666667
O-output: 1.0
F-input: 1.0
F-output: 1.0
P-input: 0.8666666666666667
P-output: 0.6666666666666666
```

همان طور که مشاهده میشود دو پترن R و P به علت شباهت زیادی که به هم دارند بیشترین خطا و کمترین دقت را در مقایسه با بقیه دارند.