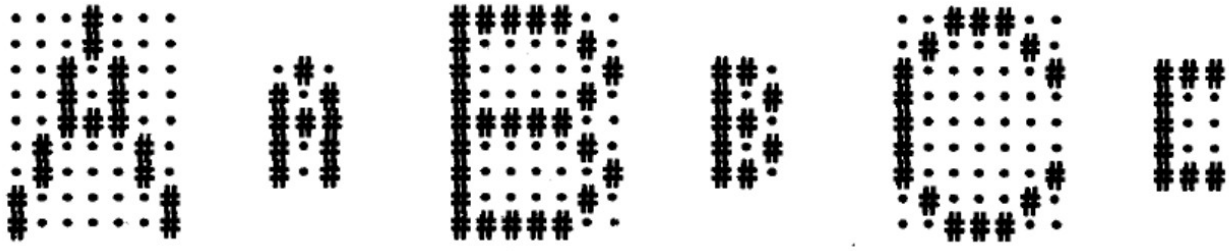
سؤال ۱)

در این سؤال می‌خواهیم با استفاده از قانون Hebb یک شبکه تک لایه درست کنیم تا این شبکه بتواند ۳ شکل ورودی خروجی را در خور ذخیره کند و در صورت داشتن نویز و یا از دست دادن اطلاعات آن آن را مورد بررسی قرار دهیم
شکل‌های ورودی ما ۳ حرف با سایز ۹*۷ و خروجی های آن‌ها با سایز ۵*۳ است که مانند شکل زیر آمده است



ابتدا ورودی و خروجی ها را رسم می کنیم:

```
input_patterns = [([0, 3], [1, 3], [2, 2], [2, 4], [3, 2], [3, 4], [4, 2], [4, 3], [4, 4], [5, 1], [5, 5], [6, 1],
                   [6, 5], [7, 0], [7, 6], [8, 0], [8, 6]),
                  ([0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [1, 0], [1, 5], [2, 0], [2, 6], [3, 0], [3, 5], [4, 0],
                   [4, 1], [4, 2], [4, 3], [4, 4], [5, 0], [5, 5], [6, 0], [6, 6], [7, 0], [7, 5], [8, 0], [8, 1],
                   [8, 2], [8, 3], [8, 4]),
                  ([0, 2], [0, 3], [0, 4], [1, 1], [1, 5], [2, 0], [2, 6], [3, 0], [4, 0], [5, 0], [6, 0], [6, 6],
                   [7, 1], [7, 5], [8, 2], [8, 3], [8, 4])]

output_patterns = [([0, 1], [1, 0], [1, 2], [2, 0], [2, 1], [2, 2], [3, 0], [3, 2], [4, 0], [4, 2]),
                   ([0, 0], [0, 1], [1, 0], [1, 2], [2, 0], [2, 1], [3, 0], [3, 2], [4, 0], [4, 1]),
                   ([0, 0], [0, 1], [0, 2], [1, 0], [2, 0], [3, 0], [4, 0], [4, 1], [4, 2])]
```

سپس با استفاده از این شکل‌ها که به صورت زیر ساخته شده اند:

```
def create_patterns(number_of_patterns, patterns, shape):
    patterns_array = []
    for i in range(number_of_patterns):
        patterns_array.append(create_new_input(shape, -1))
        create_character(patterns[i], patterns_array[i])
        # print_character(patterns_array[i])
        patterns_array[i] = patterns_array[i].flatten()
        patterns_array[i] = np.matrix(patterns_array[i])
    return patterns_array
```

سپس با استفاده از قانون Hebb ماتریس وزن را بدست می اوریم:

```
def find_weight(inputs_matrix, outputs_matrix):
    weight_matrix = np.matrix(np.zeros(shape=(63, 15)))
    for i in range(len(inputs_matrix)):
        weight_matrix = inputs_matrix[i].transpose().dot(outputs_matrix[i]) + weight_matrix
    return weight_matrix
```

و ماتریس وزن با ابعاد بعد خروجی * بعد ورودی می‌شود که در اینجا ورودی ما ۹ * ۷ یعنی ۶۳ و خروجی ۵ * ۳ یعنی ۱۵ است پس ماتریس وزن ما ۶۴ در ۱۵ می‌شود

حال اگر بخواهیم یک درصدی از کل داده‌های ورودی را دچار نویز و یا از دست دادن اطلاع کنیم از کد زیر کی توانیم استفاده کنیم :

```
def add_noise(percentage, patterns):
    output_with_noise = []
    for i in range(len(patterns)):
        temp = np.squeeze(np.asarray(copy.deepcopy(inputs[i])))
        random_choose = np.random.choice(len(temp), int((len(temp) * percentage) / 100), replace=False)
        for j in random_choose:
            temp[j] = -temp[j]
        output_with_noise.append(np.matrix(temp))
    return output_with_noise


def add_lose(percentage, patterns):
    output_with_lose = []
    for i in range(len(patterns)):
        temp = np.squeeze(np.asarray(copy.deepcopy(inputs[i])))
        random_choose = np.random.choice(len(temp), int((len(temp) * percentage) / 100), replace=False)
        for j in random_choose:
            temp[j] = 0
        output_with_lose.append(np.matrix(temp))
    return output_with_lose
```

حال شبکه را در حالت‌های مختلف ۲۰ و ۴۰ درصد نویز و از دادن اطلاعات چند بار امتحان می‌کنیم که یک بار آن به صورت زیر می شود:

```
print("Finding outputs from inputs without noise and lose: ")
compare_inputs_outputs(inputs, outputs, weight)

print("Finding outputs from inputs with 20% noise: ")
inputs_with_noise_20 = add_noise(20, inputs)
compare_inputs_outputs(inputs_with_noise_20, outputs, weight)

print("Finding outputs from inputs with 40% noise: ")
inputs_with_noise_40 = add_noise(40, inputs)
compare_inputs_outputs(inputs_with_noise_40, outputs, weight)

print("Finding outputs from inputs with 20% lose: ")
inputs_with_lose_20 = add_lose(20, inputs)
compare_inputs_outputs(inputs_with_lose_20, outputs, weight)

print("Finding outputs from inputs with 40% lose: ")
inputs_with_lose_40 = add_lose(40, inputs)
compare_inputs_outputs(inputs_with_lose_40, outputs, weight)
```

```
Finding outputs from inputs without noise and lose:
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True

Finding outputs from inputs with 20% noise:
Pattern number  0 True
Pattern number  1 True
Pattern number  2 False

Finding outputs from inputs with 40% noise:
Pattern number  0 False
Pattern number  1 True
Pattern number  2 False

Finding outputs from inputs with 20% lose:
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True

Finding outputs from inputs with 40% lose:
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True
```

برای اینکه از نتایج مطمین شویم و درصد آن را اندازه بگیریم این کار را ۱۰ بار انجام می‌دهیم و تعداد هر کدام را می نویسیم:

	Noise 20%	Noise 40%	Loss 20%	Loss 40%
Pattern A	10/10	6/10	10/10	10/10
Pattern B	10/10	2/10	10/10	10/10
Pattern C	10/10	3/10	10/10	9/10

طبق جدول بالا می‌بینیم شبکه تقریباً نسبت به از دست داده اطلاعات تا حتی ۴۰ درصد نیز مقاوم است ولی نسبت به اغتشاش به این اندازه مقاوم نیست و دیدیم تا ۲۰ درصد توانسته درست تشخیص دهد واضح است مقاومت نسبت به از دادن اطلاعات بیشتر است

حال می‌خواهیم حداکثر مقاومت شبکه را نسبت به اغتشاش بدست اوریم برای اینکار شبکه را با اغتشاش های مختلف برای ۱۰ بار اجرا می‌کنیم و برای یک بار آن داریم:

```
With   40   Percent Noise:
Pattern number   0 True
Pattern number   1 False
Pattern number   2 False

With   50   Percent Noise:
Pattern number   0 False
Pattern number   1 False
Pattern number   2 False

With   60   Percent Noise:
Pattern number   0 False
Pattern number   1 False
Pattern number   2 False

With   70   Percent Noise:
Pattern number   0 False
Pattern number   1 False
Pattern number   2 False

With   80   Percent Noise:
Pattern number   0 False
Pattern number   1 False
Pattern number   2 False

With   90   Percent Noise:
Pattern number   0 False
Pattern number   1 False
Pattern number   2 False
```

همان‌طور که دیده می‌شود شبکه برای بالای ۴۰ درصد نمی‌تواند درست تشخیص دهد ولی برای پایین‌تر از آن شبکه را برای ۱۰ بار اجرا می‌کنیم و داریم:
طبق جدول می‌بینیم که شبکه تا ۳۰ درصد را می‌تواند مقاومت کند ولی ۴۰ را نه پس حداکثر ۳۰ درصد است

	Noise 20%	Noise 30%
Pattern A	10/10	10/10
Pattern B	10/10	10/10
Pattern C	10/10	10/10

سوال ۲)

در این سؤال می‌خواهیم یک شبکه یک لایه با استفاده از قانون modified Hebb آموزش دهیم تا بتوانیم چند بردار را دز شبکه ذخیره کرد

برای این کار ابتدا بردار داده شده را با استفاده از قانون ذخیره می کنیم:

```
s = []
s0 = np.matrix([1, 1, 1, -1])
s.append(s0)
weight = find_weight(s, s)
print(weight)
```

که برای پیدا کردن ماتریس وزن مانند زیر عمل می کنیم:

```
def find_weight(inputs_matrix, outputs_matrix):
    weight_matrix = np.matrix(np.zeros(shape=(4, 4)))
    for i in range(len(inputs_matrix)):
        weight_matrix = inputs_matrix[i].transpose().dot(outputs_matrix[i]) + weight_matrix
    di = np.diag_indices(4)
    weight_matrix[di] = 0
    return weight_matrix
```

طبق این قانون بردار را در ترانهاده خودش ضرب می‌کنیم و سپس قطر اصلی را صفر می‌کنیم و ماتریس به شکل زیر می شود:

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

و سپس چک می‌کنیم که شبکه آن را ذخیره کرده است:

```
Pattern number  0 True
```

حال می‌خواهیم یک بردار مانند بردار زیر را ذخیره کنیم:

```
s1 = np.matrix([1, -1, 1, 1])
s.append(s1)
weight = find_weight(s, s)
print(weight)
compare_inputs_outputs(s, s, weight)
```

برای این کار ابتدا ماتریس وزن آن دو بردار ذخیره شده را حساب می‌کنیم و سپس چک می‌کنیم که آیا ذخیره کرده است یا نه:

```
[[ 0.  0.  2.  0.]
 [ 0.  0.  0. -2.]
 [ 2.  0.  0.  0.]
 [ 0. -2.  0.  0.]]
Pattern number  0 True
Pattern number  1 True
```

می‌دانیم این بردار را توانستیم ذخیره کنیم چرا که این بردار با بردار اولیه ما عمود است و می‌دانیم شرط کافی برای ذخیره عمود بودن بردار هاست

حال اگر بخواهیم بردار دیگه ای را به جای این بردار ذخیره کنیم مانند بردار زیر:

```
print("2 vectors are not orthogonal so we cant save it")
s1 = np.matrix([-1, 1, 1, -1])
s.append(s1)
weight = find_weight(s, s)
print(weight)
compare_inputs_outputs(s, s, weight)
```

بعد از ذخیره این بردار و چک کردن اینکه آیا ذخیره شده است یا نه می بینیم:

```
2 vectors are not orthogonal so we cant save it
[[ 0.  0.  0.  0.]
 [ 0.  0.  2. -2.]
 [ 0.  2.  0. -2.]
 [ 0. -2. -2.  0.]]
Pattern number  0 False
Pattern number  1 False
```

می‌بینیم شبکه نتوانست آن را ذخیره کند

این به این دلیل است که این بردار با بردار اولیه ما عمود نبود پس شرط کافی را نداشته است هر چند باز هم می‌تواند برداری را یافت که عمود نباشد و شبکه آن را ذخیره کند چرا که این فقط شرط کافی است ولی برای بردار جدید باید چک شود که ذخیره

شده است یا نه ولی اگر عمود باشد می‌دانیم می‌توانیم آن را ذخیره کنیم

حال می‌خواهیم به جز ۲ بردار اول بردار های دیگری را نیز در شبکه ذخیره کنیم برای این کار هر دفعه بردار را اضافه می‌کنیم ماتریس وزن را محاسبه می‌کنیم و نگاه می‌کنیم که آیا شبکه آن را درست ذخیره کرده است یا نه
می‌دانیم شرط کافی دو به دو عمود بودن بردار هاست پس چنین بردار هایی را به شبکه اضافه می کنیم:

```
s1 = np.matrix([1, -1, 1, 1])
s.append(s1)
weight = find_weight(s, s)
print(weight)
compare_inputs_outputs(s, s, weight)

s2 = np.matrix([-1, 1, 1, 1])
s.append(s2)
weight = find_weight(s, s)
print(weight)
compare_inputs_outputs(s, s, weight)

s3 = np.matrix([1, 1, -1, 1])
s.append(s3)
weight = find_weight(s, s)
print(weight)
compare_inputs_outputs(s, s, weight)
```

این سه بردار را علاوه بر بردار اول که در سؤال بود می‌خواهیم به شبکه اضافه کنیم و انهارا چک کنیم می بینیم:

```
[[ 0.  0.  2.  0.]
 [ 0.  0.  0. -2.]
 [ 2.  0.  0.  0.]
 [ 0. -2.  0.  0.]]
Pattern number  0 True
Pattern number  1 True

[[ 0. -1.  1. -1.]
 [-1.  0.  1. -1.]
 [ 1.  1.  0.  1.]
 [-1. -1.  1.  0.]]
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Pattern number  0 False
Pattern number  1 False
Pattern number  2 False
Pattern number  3 False
```

همان‌طور که میبینیم شبکه ۲ بردار اول را درست ذخیره کرده در این حالت ما ۳ بردار ذخیره شده داریم
حال اگر بخواهیم بردار ۴ ام را نیز ذخیره کنیم می‌بینیم ماتریس وزن به ۰ تبدیل می‌شود و دیگر نمی‌تواند بردار را ذخیره کند اگر هر بردار دیگه ای نیز جای آن بخواهیم ذخیره کنیم همین مشکل پیش می‌آید
پس شبکه تا ۳ بردار درست ذخیره کرده که این عدد همان یکی کمتر از تعداد بعد ورودی یا همان تعداد نورون ورودی است
این اتفاق برای بردار های عمود به هم می‌افتد اگر بردار ها عمود نباشند ممکن است بتوان باز هم ذخیره کرد ولی شرط کافی را ندارند و هر بردار جدید باید چک شود و در حالت عمود قانون یکی کمتر از تعداد نورون یعنی n-1 همیشه درست است

سؤال ۳)

در این سؤال می‌خواهیم یک شبکه عصبی تکرار شونده در خود را در دو حالت با دو الگوریتم پیاده‌سازی کنیم تا یک بردار را ذخیره کند و آن دو را در حالت اغتشاش و از دست دادن اطلاعات با هم مقایسه می‌کنیم

در این سؤال ابتدا حالت اول که iterative auto-associative است را پیاده‌سازی می‌کنیم و از در آن مانند سؤال قبل از قانون Modifeid Hebb استفاده می‌کنیم برای این کار مانند زیر عمل می کنیم:

```
def iterative_net(inputs_pattern, tests_pattern):
    weight = find_weight(inputs_pattern, inputs_pattern)
    print(weight, "\n")
    last_result = []

    for i in range(len(tests_pattern)):
        result_patterns = []
        new_pattern = np.sign(tests_pattern[i] * weight)
        while not np.all(new_pattern == inputs_pattern):
            result_patterns.append(np.array(new_pattern).reshape(4,))
            new_pattern = np.sign(new_pattern * weight)
            if check_repeated_results(result_patterns, new_pattern):
                print("Repeated pattern for Pattern: ", i)
                break
        last_result.append(np.array(new_pattern).reshape(4,))
    print(last_result)
    compare_inputs_outputs(np.array(last_result), inputs_pattern, weight)
```

که در آن ابتدا با استفاده از بردار ورودی ماتریس وزن را با استفاده از قانون Modified Hebb بدست می‌آوریم سپس به ازای هر ورودی که برای تست در شبکه داده‌ایم که همان تغییر یافته ورودی اصلی است ورودی را در ماتریس ضرب می‌کنیم و شرط توقف را چک می‌کنیم

شرط توقف در آن این است که بردار بدست آمده یا برابر بردار ورودی شود که یعنی در این صورت شبکه به درستی ورودی تعییر یافته را تشخیص داده و یا به یک بردار که قبلاً به آن رسیده است برسد که یعنی دیگر نمی‌تواند به بردار ورودی برسد

در این نوع شبکه‌ها بردار تغییر یافته در ماتریس وزن ضرب شده و دوباره به شبکه داده می‌شود این کار را آن‌قدر تکرار می‌کنیم تا

به شرط توقف برسیم و در آخر بردار های بدست آورده شده را مقایسه می‌کنیم
حال بردار ورودی را دچار اغتشاش و از دادن اطلاعات می‌کنیم مانند شکل زیر:

```
def lose_3_value():
    pattern = []
    pattern.append(np.matrix([0, 0, 0, -1]))
    pattern.append(np.matrix([0, 0, 1, 0]))
    pattern.append(np.matrix([0, 1, 0, 0]))
    pattern.append(np.matrix([1, 0, 0, 0]))
    return pattern


def noise_3_value():
    pattern = []
    pattern.append(np.matrix([1, -1, -1, 1]))
    pattern.append(np.matrix([-1, 1, -1, 1]))
    pattern.append(np.matrix([-1, -1, 1, 1]))
    pattern.append(np.matrix([-1, -1, -1, -1]))
    return pattern
```

و این بردار ها را به شبکه می‌دهیم تا آن‌ها را پیش‌بینی کند و داریم:

```
s = []
s0 = [1, 1, 1, -1]
s.append(np.matrix([1, 1, 1, -1]))

iterative_net(np.matrix(s0), lose_3_value())
iterative_net(np.matrix(s0), noise_3_value())
```

و در خروجی داریم:
برای حالتی که ورودی ۷۵٪ اطلاعات خودش را از دست بدهد داریم:

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

[array([ 1.,  1.,  1., -1.]), array([ 1.,  1.,  1., -1.]), array([ 1.,  1.,  1., -1.]), array([ 1.,  1.,  1., -1.])]
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True
Pattern number  3 True
```

می‌بینیم که شبکه در این حالت همه ی حالت‌ها را درست تشخیص داده است
حال برای حالتی که شبکه ورودی ۷۵٪ اغتشاش دارد داریم:

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

Repeated pattern for Pattern:  0
Repeated pattern for Pattern:  1
Repeated pattern for Pattern:  2
Repeated pattern for Pattern:  3
[array([-1., -1., -1.,  1.]), array([-1., -1., -1.,  1.]), array([-1., -1., -1.,  1.]), array([-1., -1., -1.,  1.])]
Pattern number  0 False
Pattern number  1 False
Pattern number  2 False
Pattern number  3 False
```

می‌بینیم برای همه حالت‌ها شبکه به تکرار رسیده است و نتوانسته ورودی اصلی را پیدا کند

حال همین کار ها را می‌خواهیم با شبکه Hobfeild که نوعی شبکه تکرار شونده است انجام دهیم
در این شبکه یک ابدیت به صورت ای که هر سری یک عضو از بردار اپدیت می‌شود انجام می‌شود
به این منظور که هر سری یک عضو از بردار تست را در ستون مربوط به آن در ماتریس وزن ضرب می‌کنیم آن عضو را اپدیت می‌کنیم و دوباره بردار را به شبکه می‌دهیم این کار را انقدر تکرار می‌کنیم تا در ۲ اپدیت متوالی تغییری ایجاد نشود و به یک بردار برسیم حال این بردار ممکن است بردار درست ما یعنی همان بردار اصلی باشد و یا یک بردار غلط ولی اثبات می‌شود که این شبکه همیشه همگرا است و اپدیت کردن وزن ها را باید به صورت رندوم اعمال کنیم تا همگرا شود

شبکه را مانند زیر می سازیم:

```
def hopfeild_net(inputs_pattern, tests_pattern):
    weight = find_weight(inputs_pattern, inputs_pattern)
    weight = np.array(weight)
    print(weight, "\n")
    last_result = []
    for i in range(len(tests_pattern)):
        x = np.array(tests_pattern[i]).reshape(4, )
        y = x
        indx = create_random_index(len(y))
        for j in indx:
            y[j] = np.sign(x[j] + np.sum(y * weight.transpose()[j]))
        last_result.append(y)
    print(last_result)
    compare_inputs_outputs(np.matrix(last_result), inputs_pattern, weight)
```

که در آن بعد از پیدا کردن ماتریس وزن برای هر ورودی تست انرا برابر با x , y می‌گذاریم و هر سری به صورت رندوم یک عضو از y را انتخاب کرده و اپدیت می‌کنیم و در آخر خروجی را بعد از چک کردن شرط توقف مقایسه می‌کنیم

برای آن مانند حالت قبل داریم:

```
s = []
s0 = [1, 1, 1, -1]
s.append(np.matrix([1, 1, 1, -1]))

# iterative_net(np.matrix(s0), lose_3_value())
# iterative_net(np.matrix(s0), noise_3_value())

hopfeild_net(np.matrix(s0), lose_3_value())
hopfeild_net(np.matrix(s0), noise_3_value())
```

و در خروجی داریم:

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

[array([ 1,  1,  1, -1]), array([ 1,  1,  1, -1]), array([ 1,  1,  1, -1]), array([ 1,  1,  1, -1])]
Pattern number  0 True
Pattern number  1 True
Pattern number  2 True
Pattern number  3 True

[[ 0.  1.  1. -1.]
 [ 1.  0.  1. -1.]
 [ 1.  1.  0. -1.]
 [-1. -1. -1.  0.]]

[array([-1, -1, -1,  1]), array([-1, -1, -1,  1]), array([-1, -1, -1,  1]), array([-1, -1, -1,  1])]
Pattern number  0 False
Pattern number  1 False
Pattern number  2 False
Pattern number  3 False
```

می‌بینیم که شبکه در برابر از دست دادن اطلاعات درست عمل کرده است ولی در حالت وجود اغتشاش نتوانسته به دردستی ورودی را تشخیص دهد

در شبکه Hobfeil میزان تعداد بردار ذخیره شده تقریباً برابر با

$$P \approx \frac{n}{2 \log_2 n}.$$

است که در آن n همان تعداد نورون یا بعد ورودی است که در اینجا بردار ما ۴ بعدی بود پس تعداد برداری که می‌تواند ذخیره کند برابر با ۱ می‌شود
با اینکه در اینجا در هر دو شبکه نتوانستیم در حالت اغتشاش به بردار ورودی برسیم ولی می‌دانیم شبکه hobfeil اثبات دارد که همیشه همگرا می‌شود ولی در حالت اول همچین چیزی وجود ندارد و ممکن است همگرا نشود و ما به یک بردار واحد نرسیم پس شبکه hobfeil برای این کار بهتر است

سؤال ۴)

در این سؤال می‌خواهیم با استفاده از یک شبکه حافظه دار دوطرفه ۲ الگوی ورودی-خروجی را ذخیره کنیم
در این شبکه همیشه می‌توان با استفاده از الگوی خروجی به ورودی و با استفاده از خروجی به ورودی رسید

در اینجا ابتدا الگو های ورودی و خروجی را مانند زیر درست می کنیم:

```
x_layer = [([0, 7], [0, 8], [1, 7], [1, 8], [2, 7], [2, 8], [3, 7], [3, 8], [4, 7], [4, 8], [5, 6], [5, 7], [5, 8],
            [5, 9], [6, 5], [6, 6], [6, 7], [6, 8], [6, 9], [6, 10], [7, 4], [7, 5], [7, 6], [7, 7], [7, 8], [7, 9],
            [7, 10], [7, 11], [8, 3], [8, 4], [8, 5], [8, 6], [8, 7], [8, 8], [8, 9], [8, 10], [8, 11], [8, 12], [9, 2],
            [9, 3], [9, 4], [9, 7], [9, 8], [9, 11], [9, 12], [9, 13], [10, 1], [10, 2], [10, 3], [10, 7], [10, 8],
            [10, 12], [10, 13], [10, 14], [11, 0], [11, 1], [11, 2], [11, 7], [11, 8], [11, 13], [11, 14], [11, 15],
            [12, 0], [12, 1], [12, 7], [12, 8], [12, 14], [12, 15], [13, 0], [13, 7], [13, 8], [13, 15], [14, 7],
            [14, 8], [15, 6], [15, 7], [15, 8], [15, 9], [16, 6], [16, 7], [16, 8], [16, 9], [17, 6], [17, 9]),
           ([0, 6], [0, 7], [0, 8], [0, 9], [1, 7], [1, 8], [2, 7], [2, 8], [3, 7], [3, 8], [4, 7], [4, 8], [5, 7],
            [5, 8], [6, 7], [6, 8], [7, 7], [7, 8], [8, 7], [8, 8], [9, 2], [9, 3], [9, 7], [9, 8], [9, 12], [9, 13],
            [10, 2], [10, 3], [10, 6], [10, 7], [10, 8], [10, 9], [10, 12], [10, 13], [11, 2], [11, 3], [11, 5],
            [11, 6],
            [11, 7], [11, 8], [11, 9], [11, 10], [11, 12], [11, 13], [12, 2], [12, 3], [12, 4], [12, 5], [12, 6],
            [12, 7], [12, 8], [12, 9], [12, 10], [12, 11], [12, 12], [12, 13], [13, 2], [13, 3], [13, 4], [13, 5],
            [13, 6], [13, 7], [13, 8], [13, 9], [13, 10], [13, 11], [13, 12], [13, 13], [14, 2], [14, 3], [14, 5],
            [14, 6], [14, 7], [14, 8], [14, 9], [14, 10], [14, 12], [14, 13], [15, 2], [15, 3], [15, 6], [15, 7],
            [15, 8], [15, 9], [15, 12], [15, 13], [16, 2], [16, 3], [16, 12], [16, 13], [17, 2], [17, 3], [17, 12],
            [17, 13])]
```

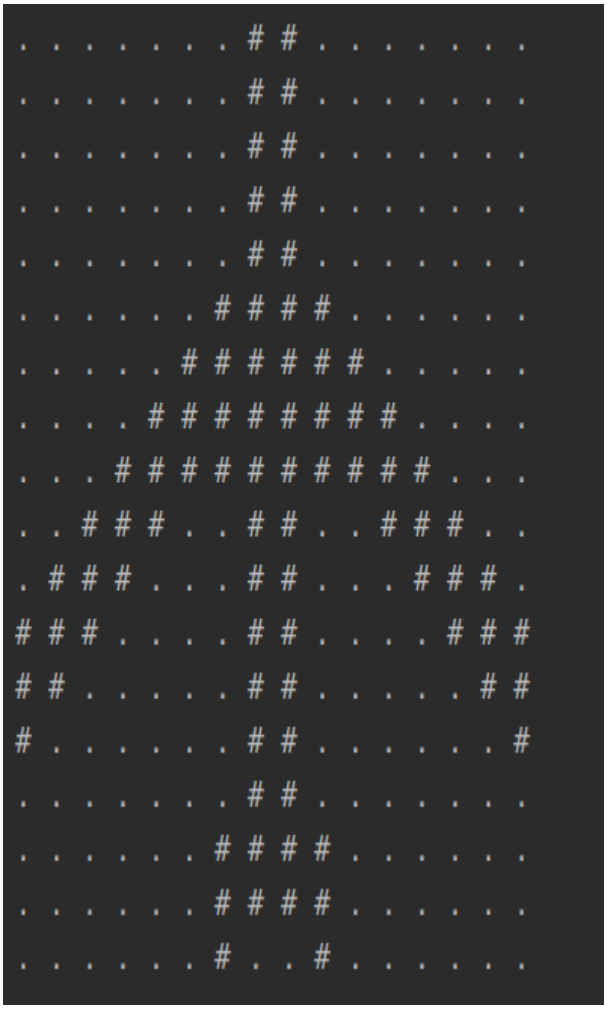
```
y_layer = [([0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 7], [0, 16], [0, 17], [0, 21], [0, 26], [0, 28], [0, 29],
            [0, 31], [0, 32], [0, 33], [1, 0], [1, 5], [1, 7], [1, 15], [1, 18], [1, 21], [1, 22], [1, 26], [1, 28],
            [2, 0], [2, 5], [2, 7], [2, 15], [2, 18], [2, 21], [2, 22], [2, 26], [2, 28], [3, 0], [3, 5], [3, 7],
            [3, 15], [3, 18], [3, 21], [3, 23], [3, 26], [3, 28], [3, 29], [3, 30], [3, 31], [3, 32], [4, 0], [4, 1],
            [4, 2], [4, 3], [4, 4], [4, 7], [4, 14], [4, 19], [4, 21], [4, 24], [4, 26], [4, 28], [5, 0], [5, 7],
            [5, 14], [5, 15], [5, 16], [5, 17], [5, 18], [5, 19], [5, 21], [5, 25], [5, 26], [5, 28], [6, 0], [6, 7],
            [6, 12], [6, 14], [6, 19], [6, 21], [6, 25], [6, 26], [6, 28], [7, 0], [7, 7], [7, 8], [7, 9], [7, 10],
            [7, 11], [7, 12], [7, 14], [7, 19], [7, 21], [7, 26], [7, 28], [7, 29], [7, 30], [7, 31], [7, 32],
            [7, 33]),
           ([0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 12], [0, 18], [0, 25], [0, 28], [0, 33],
            [1, 3], [1, 11], [1, 13], [1, 18], [1, 19], [1, 25], [1, 28], [1, 32], [2, 3], [2, 10], [2, 14], [2, 18],
            [2, 20], [2, 25], [2, 28], [2, 31], [3, 3], [3, 10], [3, 14], [3, 18], [3, 21], [3, 25], [3, 28], [3, 29],
            [3, 30], [4, 3], [4, 9], [4, 15], [4, 18], [4, 22], [4, 25], [4, 28], [4, 31], [5, 3], [5, 9], [5, 10],
            [5, 11], [5, 12], [5, 13], [5, 14], [5, 15], [5, 18], [5, 23], [5, 25], [5, 28], [5, 32], [6, 3], [6, 9],
            [6, 15], [6, 18], [6, 24], [6, 25], [6, 28], [6, 33], [7, 3], [7, 9], [7, 15], [7, 18], [7, 25], [7, 28],
            [7, 34])]
```

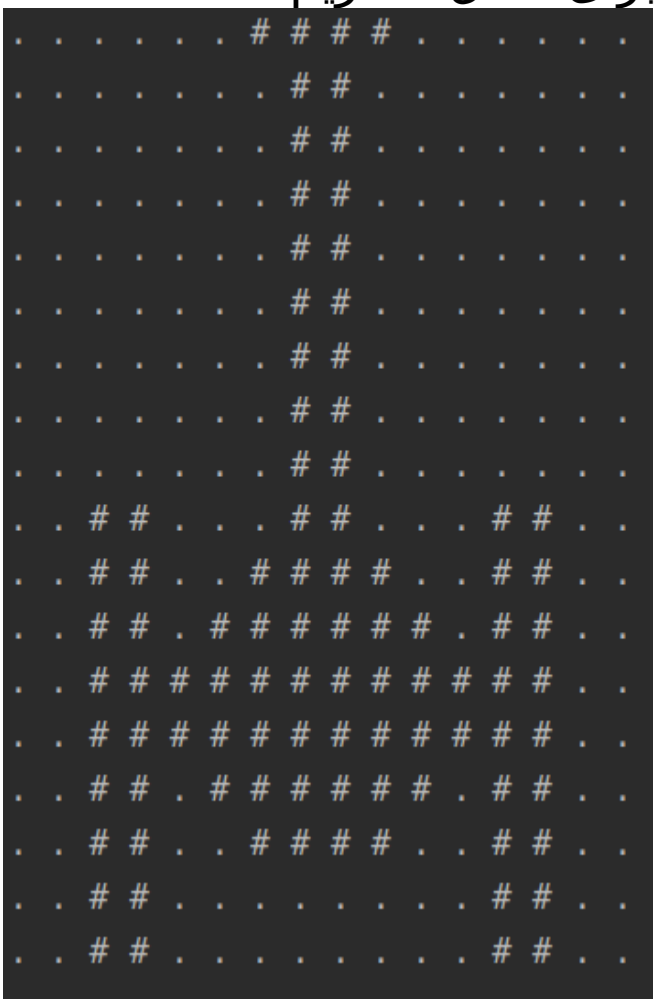
سپس با استفاده از آن‌ها یک الگو می سازیم:

```
def create_new_input(shape, value):
    array = np.empty(shape)
    array.fill(value)
    return array


def create_character(values, array):
    for i, j in values:
        array[i][j] = 1
```

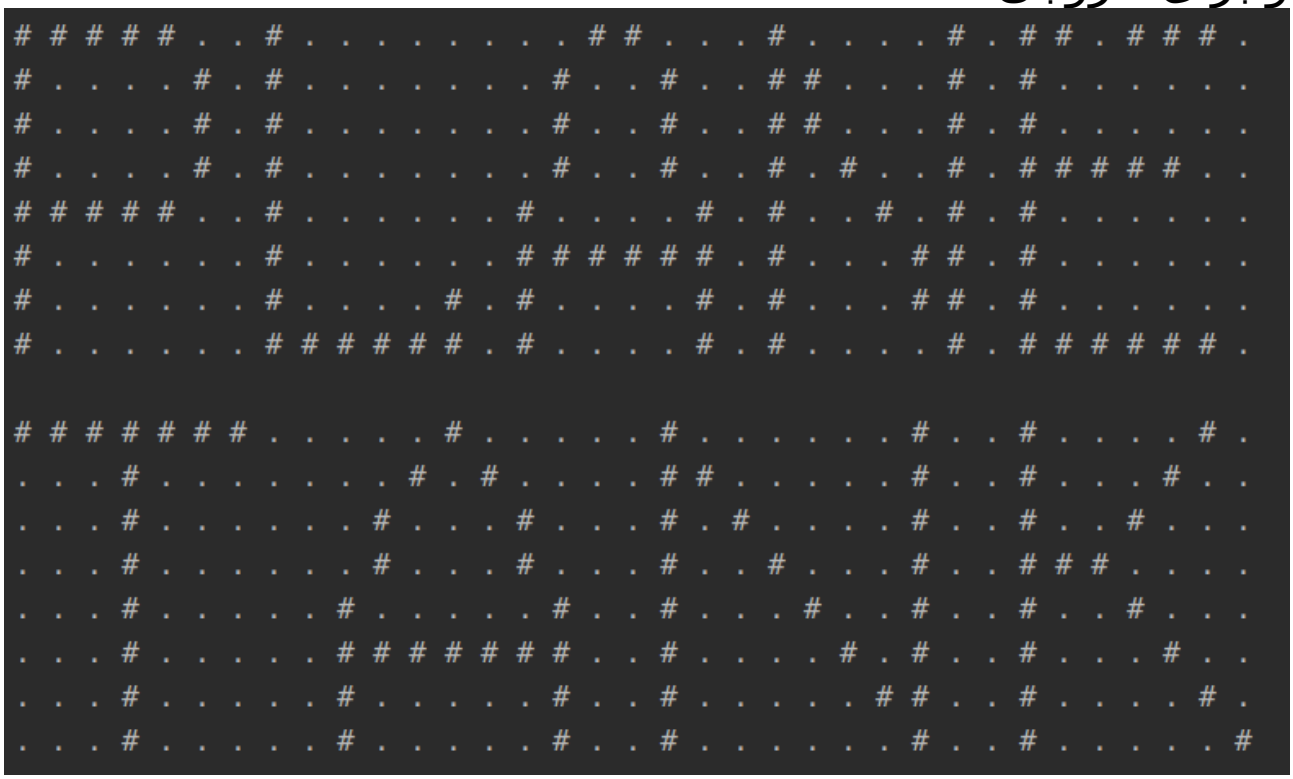
```python
def create_patterns(number_of_patterns, patterns, shape):
    patterns_array = []
    for i in range(number_of_patterns):
        patterns_array.append(create_new_input(shape, -1))
        create_character(patterns[i], patterns_array[i])
        print_character(patterns_array[i])
        patterns_array[i] = patterns_array[i].flatten()
    return patterns_array
```

و سپس بعد از نشان دادن آن‌ها به یک ارایه ی یک بعدی به تعداد ضرب ابعاد آن درست می‌کنیم
برای شکل‌ها داریم:





و برای خروجی:



حال با استفاده از این الگو ها ماتریس وزن برای این شبکه را بدست می اوریم:

```
def find_weight(inputs_matrix, outputs_matrix):
    weight_matrix = np.array(np.zeros(shape=(288, 280)))
    for i in range(len(inputs_matrix)):
        weight_matrix = np.matrix(inputs_matrix[i]).transpose().dot(np.matrix(outputs_matrix[i])) + np.matrix(weight_matrix)
    return weight_matrix
```

که در این شبکه یک ماتریس با ابعداد ۲۸۸*۲۸۰ است و به صورت زیر می شود:

```
[[-2. -2. -2. ...  0.  0.  0.]
 [-2. -2. -2. ...  0.  0.  0.]
 [-2. -2. -2. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ... -2. -2.  2.]
 [-2. -2. -2. ...  0.  0.  0.]
 [-2. -2. -2. ...  0.  0.  0.]]
```

حال می‌توانیم شبکه را امتحان کنیم
برای اینکار یک بار با استفاده از ورودی و ضرب آن در ماتریس وزن به الگو های خروجی می‌رسیم
و بار دیگر با ضرب ماتریس خروجی در ترانهاده ماتریس وزن باید به ورودی برسیم که در این شبکه داریم:

```python
def compare_inputs_outputs_x_to_y(inputs_array, outputs_array, weight_array):
    for i in range(len(inputs_array)):
        print("Pattern number ", i, end=" ")
        if np.all(np.sign(inputs_array[i] * weight_array) == outputs_array[i]):
            print("True")
        else:
            print("False")
    print()


def compare_inputs_outputs_y_to_x(inputs_array, outputs_array, weight_array):
    for i in range(len(inputs_array)):
        print("Pattern number ", i, end=" ")
        if np.all(np.sign(np.matrix(outputs_array[i]) * np.array(weight_array).transpose()) == inputs_array[i]):
            print("True")
        else:
            print("False")
    print()
```

و برای اجرای آن در شبکه داریم:

```python
print("Compare True Inputs to True Outputs to check the model")
print("Getting outputs from inputs:")
compare_inputs_outputs_x_to_y(inputs, outputs, weight)
print("Getting inputs from outputs:")
compare_inputs_outputs_y_to_x(inputs, outputs, weight)
print()
```

که در خروجی می بینیم:

```
Compare True Inputs to True Outputs to check the model
Getting outputs from inputs:
Pattern number  0 True
Pattern number  1 True

Getting inputs from outputs:
Pattern number  0 True
Pattern number  1 True
```

می‌بینیم که شبکه در هر دو حالت و برای هر دو زوج الگو درست عمل کرده است

که اگر شکل‌های آن را نشان دهیم میبینیم دقیقاً همان شکل‌های اولیه ما هستند

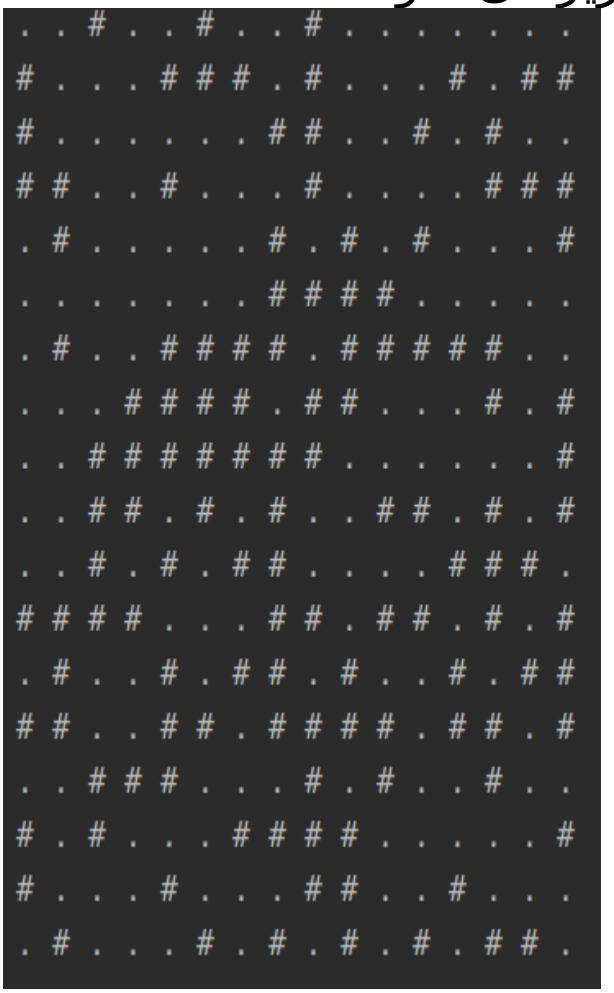
حال در قسمت بعد می‌خواهیم در شکل‌ها نویز ایجاد کنیم و با شکل‌های جدید عمل‌کرد شبکه را ببینیم
برای ایجاد نویز چون می‌خواهیم hamming شبکه نزدیک به ۳۰ باشد یعنی باید ۳۰ درصد ارایه ی شکل جدید با شکل قبل فرق کند برای این کار از تابع زیر استفاده می کنیم:

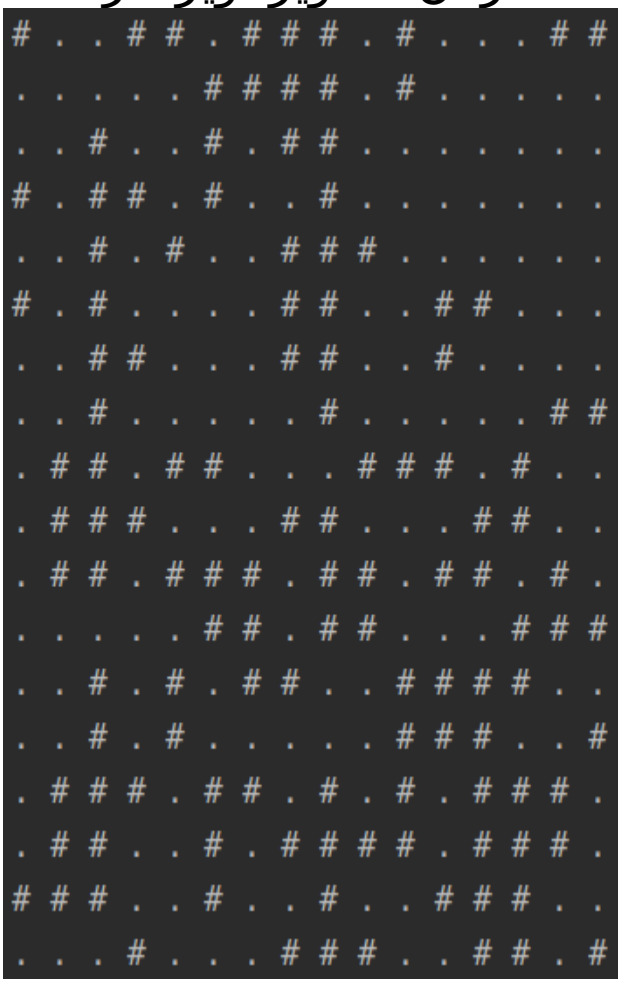
```
def noise_inputs_outputs(inputs_pattern, percent):
    output_noise = []
    for i in range(len(inputs_pattern)):
        noise_pattern = copy.deepcopy(inputs_pattern[i])
        for j in range(len(noise_pattern)):
            a = np.random.randint(1, 100)
            if a < percent:
                noise_pattern[j] = -noise_pattern[j]
        output_noise.append(noise_pattern)
    return output_noise
```

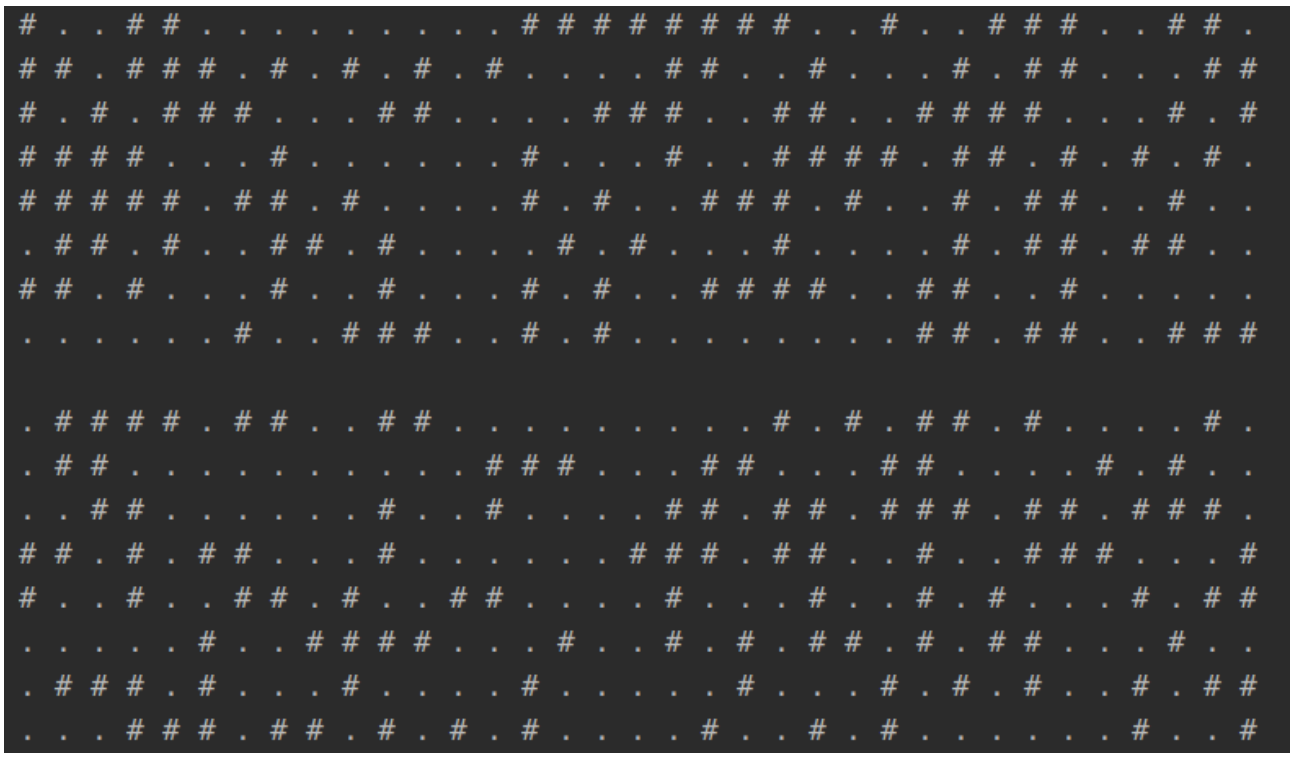
که در آن با درصدی که میدهیم مقادیر ارایه را قرینه می‌کند چرا که ارایه ما bipolar است
حال این تصاویر جدید را می‌توانیم مانند زیر بکشیم:

```
test_patterns_x = noise_inputs_outputs(inputs, 30)
test_patterns_y = noise_inputs_outputs(outputs, 30)
print_character(np.array(test_patterns_x[0]).reshape(18, 16))
print_character(np.array(test_patterns_x[1]).reshape(18, 16))
print_character(np.array(test_patterns_y[0]).reshape(8, 35))
print_character(np.array(test_patterns_y[1]).reshape(8, 35))
```

که در آن تصاویر نویز دار مانند شکل زیر می شوند:







همان‌طور که دیده می‌شود تصاویر در این حالت به اندازه ۳۰ درصد نویز دارند و با تصاویر اصلی فاصله دارند

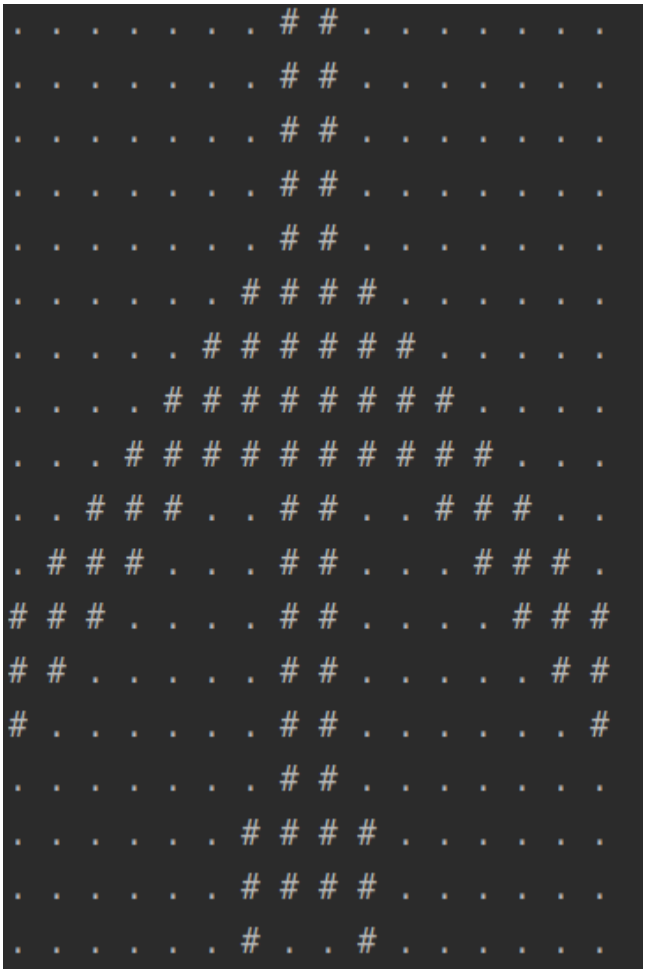
حال شبکه را پیاده‌سازی می‌کنیم تا هر سری ابتدا خروجی را از ضرب ماتریس نویز دار ورودی در ماتریس وزن و گزراندن از اکتیویشن که در اینجا همان تابع ساین است برای هر عضو خروجی بدست می‌آوریم
در این صورت به یک ماتریس خروجی می‌رسیم
حال اگر این ماتریس را در ورودی شبکه برگردانده یعنی به صورت بازگشتی عمل کنیم و آن را در ماتریس وزن برای تک تک اعضای ورودی حساب کنیم به ماتریس ورودی می‌رسیم
این کار را انقدر تکرار می‌کنیم تا ماتریس های ورودی و خروچجی دیگر تغییر نکنند این کار را مانند زیر عمل می کنیم:

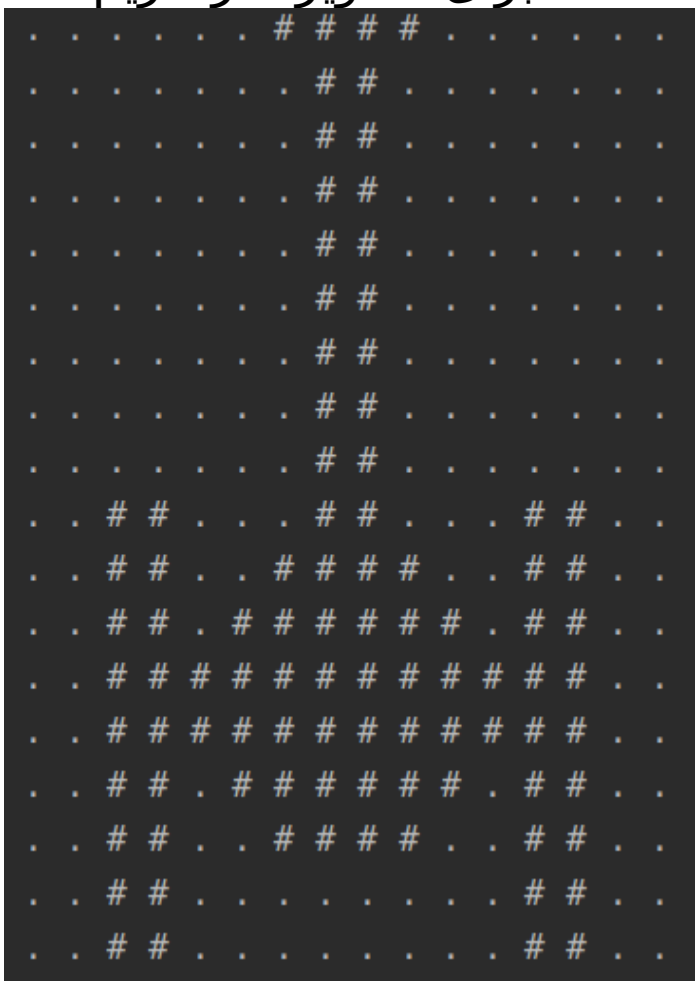
```
for i in range(len(test_patterns_x)):
    x = copy.deepcopy(list(test_patterns_x[i]))
    y = copy.deepcopy(list(test_patterns_y[i]))

    for j in range(len(y)):
        y[j] = activation(np.sum(x * np.array(weight).transpose()[j]), j, x)
    last_result_y.append(y)

    for j in range(len(x)):
        x[j] = np.sign(np.sum(np.matrix(y) * weight[j].transpose()))
    last_result_x.append(x)
```

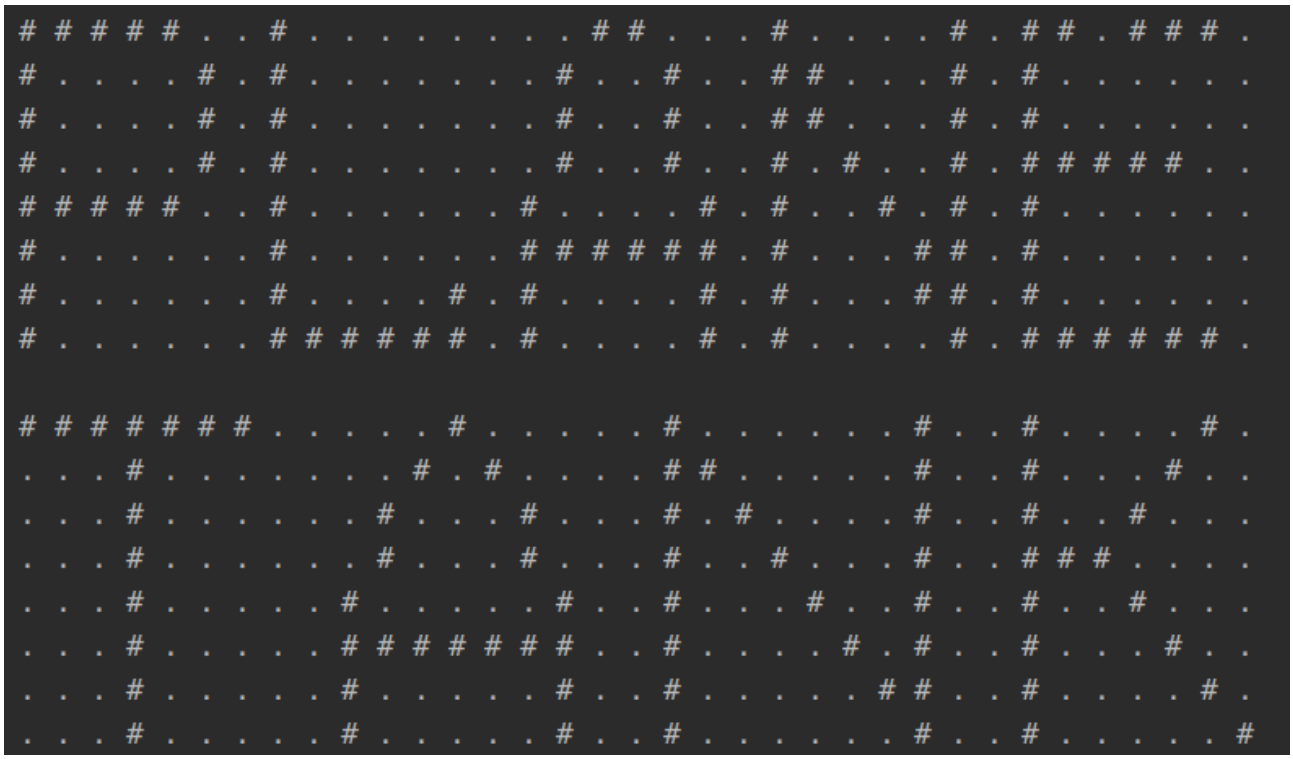
حال می توانییم شکل‌های بدست آمده را ببینیم و با ورودی اولیه شبکه مقایسه کنیم که آیا شبکه توانسته در صورتی که ۳۰ درصد نویز وجود دارد به تصویر درست برسد یا خیر:

```
print("Getting outputs from inputs:")
compare_inputs_outputs_x_to_y(last_result_x, outputs, weight)
print("Getting inputs from outputs:")
compare_inputs_outputs_y_to_x(inputs, last_result_y, weight)
show_character(np.array(last_result_x[0]).reshape(18, 16))
show_character(np.array(last_result_x[1]).reshape(18, 16))
show_character(np.array(last_result_y[0]).reshape(8, 35))
show_character(np.array(last_result_y[1]).reshape(8, 35))
```

که برای تصاویر آخر داریم:







و بعد از مقایسه تصاویر داریم:

```
Getting outputs from inputs:
Pattern number  0 True
Pattern number  1 True

Getting inputs from outputs:
Pattern number  0 True
Pattern number  1 True
```

همان‌طور که میبینیم شبکه به درستی تصاویر را بدست آورده است

شبکه این کار را با پیدا کردن ماتریس خروجی و سپس دادن آن ماتریس به ورودی و تکرار اینکار می‌تواند تصویر نویز دار را به تصاویر اصلی خود بازگرداند