به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

شبکه های عصبی

تمرین سوم

نام و نام خانوادگی	مهسا تاجیک
شماره دانشجویی	810198126
تاریخ ارسال گزارش	99/2/23

سوال 1 – Character Recognition using Hebbian Learning Rule

1) ورودی شبکه یک آرایه دوبعدی 7*9 و خروجی 3*5 است که آن ها را بصورت بردارهای 63 تایی و 15 تایی تعریف و مقداردهی کردیم :

```
import numpy as np
import matplotlib.pyplot as plt
"""""""""   initialization  """""
A = -np.ones((63, 1))
B = -np.ones((63, 1))
C = -np.ones((63, 1))
A[3] = 1
A[10] = 1
A[16] = 1
A[18] = 1
A[23] = 1
A[25] = 1
A[30] = 1
A[31] = 1
A[32] = 1
A[36] = 1
A[40] = 1
A[43] = 1
A[47] = 1
A[49] = 1
A[55] = 1
A[56] = 1
A[62] = 1
```

```
B[0] = 1
B[1] = 1
B[2] = 1
B[3] = 1
B[4] = 1
B[7] = 1
B[12] = 1
B[14] = 1
B[20] = 1
B[21] = 1
B[26] = 1
B[28] = 1
B[29] = 1
B[30] = 1
B[31] = 1
B[32] = 1
B[35] = 1
B[40] = 1
B[42] = 1
B[48] = 1
B[49] = 1
B[54] = 1
B[56] = 1
B[57] = 1
B[58] = 1
```

```
B[58] = 1
B[59] = 1
B[60] = 1

C[2] = 1
C[3] = 1
C[4] = 1
C[8] = 1
C[12] = 1
C[14] = 1
C[20] = 1
C[21] = 1
C[28] = 1
C[35] = 1
C[42] = 1
C[48] = 1
C[50] = 1
C[54] = 1
C[58] = 1
C[59] = 1
C[60] = 1
tA = np.ones((15, 1))
tB = np.ones((15, 1))
tC = np.ones((15, 1))
```

```
tA = np.ones((15, 1))
tB = np.ones((15, 1))
tC = np.ones((15, 1))

tA[0] = -1
tA[2] = -1
tA[4] = -1
tA[10] = -1
tA[13] = -1

tB[2] = -1
tB[4] = -1
tB[8] = -1
tB[10] = -1
tB[14] = -1

tC[4] = -1
tC[5] = -1
tC[7] = -1
tC[8] = -1
tC[10] = -1
tC[11] = -1
```

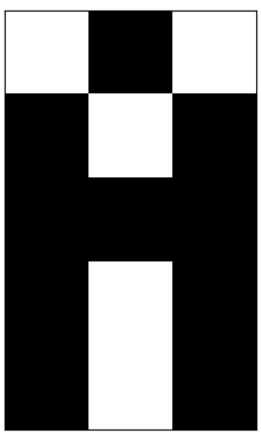
سپس بردار وزن را تعریف و مقداردهی می کنیم :

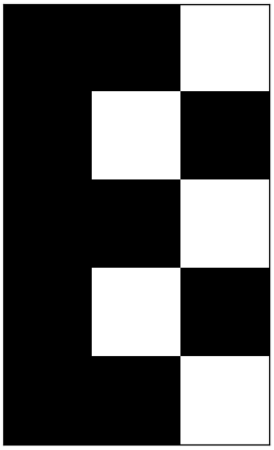
```
W = np.zeros((63, 15))
W = np.matmul(A, tA.T) + np.matmul(B, tB.T) + np.matmul(C, tC.T)
```

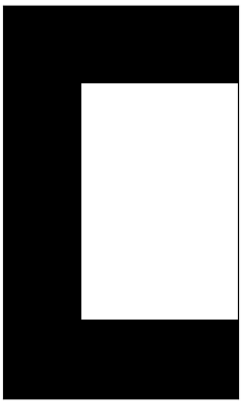
و با استفاده از بردار وزن و ورودی ها ، خروجی را تولید می کنیم :

```
yA = np.matmul(A.T, W)
yB = np.matmul(B.T, W)
yC = np.matmul(C.T, W)
fA = np.sign(yA)
fB = np.sign(yB)
fC = np.sign(yC)
```

با استفاده از کتابخانه matplotlib خروجی ها را ابتدا بصورت آرایه دوبعدی 3*5 تبدیل می کنیم و نمایش می دهیم:







همانطور که میبینیم شبکه توانست تمام ورودی ها را به خروجی مطلوب برساند.

2) در این قسمت می خواهیم با اضافه کردن 20 و 40 درصد نویز به داده ها بصورت تصادفی ، درصد تشخیص درست خروجی را بدست آوریم.

برای اضافه کردن نویز به داده ها ابتدا تابعی بنام mistake نوشتیم که یک لوپ روی داده های ورودی میزند و هرداده را با احتمال داده شده در 1- ضرب می کند.

کد این قسمت را در شکل 1-1 می بینیم :

```
def decision(probability):
    return random.random() < probability


def mistake(input, probability):
    new_input = np.copy(input)
    for i in range(len(input)):
        if decision(probability) is True:
            new_input[i] = -new_input[i]
    return new_input
```

شکل 1-1

سپس تابعی نوشتیم که از بین سه ورودی A,B,C هربار یکی را به تصادف انتخاب میکند تا آن را نویزی کنیم و به شبکه بدهیم:

```
def choose_random_input(A, B, C):
    input = np.array([A, B, C])
    random_input = np.random.choice(input.shape[0], 1)
    return random_input
```

100 بار این برنامه را تکرار می کنیم . هربار یکی از ورودی ها انتخاب می شود و به تابع mistake داده می شود و خروجی این تابع که ورودی نویزدار است به شبکه داده می شود و با استفاده از بردار وزنی که در قسمت قبل محاسبه کردیم خروجی جدید را تولید می کنیم و با خروجی قسمت قبل مقایسه می کنیم و تعداد دفعاتی که این دو خروجی یکسان هستند و شبکه به درستی قادر به بازیابی ورودی بوده است ، شمارش می کنیم.

```
detect_from_mistakes = 0
for iter1 in range(100):
    rand_in = choose_random_input(A, B, C)
    if (rand_in == 0):
        temp_A = A
        temp_A = mistake(temp_A, 0.2)
        yA_mistake = np.matmul(temp_A.T, W)
        f_mistake = np.sign(yA_mistake)
        f = fA
    elif (rand_in == 1):
        temp_B = B
        temp_B = mistake(temp_B, 0.2)
        yB_mistake = np.matmul(temp_B.T, W)
        f_mistake = np.sign(yB_mistake)
        f = fB
    elif (rand_in == 2):
        temp_C = C
        temp_C = mistake(temp_C, 0.2)
        yC_mistake = np.matmul(temp_C.T, W)
        f_mistake = np.sign(yC_mistake)
        f = fC

    if (np.array_equal(f_mistake, f)):
        detect_from_mistakes += 1
print("percentage is :", 0.2)
print("percentage of detect from mistakes :", detect_from_mistakes / 100)
```

زمانیکه 20 درصد نویز به داده ها اضافه می شود درصد دفعاتی که شبکه بدرستی خروجی ها را تولید میکند عبارتست از :

```
percentage is : 0.2
percentage of detect from mistakes : 0.97
```

این مقدار برای زمانیکه 40 درصد داده ها را نویزی می کنیم برابر است با :

```
percentage is : 0.4
percentage of detect from mistakes : 0.65
```

3) در این قسمت می خواهیم با حذف 20 و 40 درصد از داده های ورودی ، درصد تشخیص درست خروجی را بدست آوریم.

تابعی بنام miss مشابه تابع mistake تعریف می کنیم با این تفاوت که ایندکس هایی که احتمال تغییر کردن را دارند ، صفر میکنیم. بقیه مراحل مشابه حالت قبل است.

نتیجه ی حاصل از حذف 20 درصد از داده ها :

```
percentage is : 0.2
percentage of detect from misses : 1.0
```

نتیجه ی حاصل از حذف 40 درصد از داده ها :

```
percentage is : 0.4
percentage of detect from misses : 0.99
```

4) همانطور که می بینیم مقاومت شبکه در مقابل از دست رفتن اطلاعات بالاتر است و بهتر می تواند ورودی را بازیابی کند.

5)

مقاومت شبکه در برابر نویز 30 درصدی :

```
percentage is : 0.3
percentage of detect from mistakes : 0.84
```

مقاومت شبکه در برابر نویز 40 درصدی :

```
percentage is : 0.4
percentage of detect from mistakes : 0.65
```

مقاومت شبکه در برابر نویز 50 درصدی :

```
percentage is : 0.5
percentage of detect from mistakes : 0.0
```

همانطور که قابل مشاهده است با واردکردن 50 درصد نویز به داده دیگر شبکه قادر به بازیابی نخواهد بود و حداکثر مقاومت آن در برابر نویز 40 درصدی است.

سوال2- Storage Capacity in an Auto-associative Net

1) در این قسمت خواسته شده است با استفاده از قانون یادگیری modified hebbian بردار s = [1,1,1,-1] را ذخیره کنیم. کد این قسمت را در شکل 2-1 می بینیم و بعد از ذخیره ی بردار s ، ماتریس وزن و خروجی محاسبه شده است و نتیجه در شکل 2-2 آورده شده است.

```python
import numpy as np

s = np.array([[1, 1, 1, -1]])
t = np.array([[1, 1, 1, -1]])
''''''''''part 1 '''''''''
w = np.matmul(s.T, s)
w = w - np.eye(w.shape[0])
y = np.sign(np.matmul(s, w))
print("weights : \n", w)
print("output : ", y)
```

شکل 2-1

```
weights :  [[ 0.  1.  1. -1.]
 [ 1.  0.  1. -1.]
 [ 1.  1.  0. -1.]
 [-1. -1. -1.  0.]]
output :  [[ 1  1  1 -1]]
```

شکل 2-2

همانطور که انتظار داشتیم ، خروجی بدست آمده با بردار ورودی ذخیره شده یکسان است.

2) در این قسمت خواسته شده بردار دیگری را در شبکه ذخیره کنیم که قادر به بازیابی آن باشد.

بردار sample1 = [-1,1,-1,-1] را در شبکه ذخیره می کنیم. سپس مشابه قسمت قبل بردار وزن و خروجی را برای آن محاسبه می کنیم و می بینیم که خروجی برابر است با بردار [-1,1,-1,-1] و ورودی را برای ما تداعی می کند . برای اینکه شبکه قادر به بازیابی الگوی ذخیره شده ی قبلی هم باشد باید ماتریس وزن های بدست آمده را باهم جمع کرده و بروزرسانی کنیم.

کد مربوط به این قسمت را در شکل 2-3 میبینیم.

```
sample1 = np.array([[-1, 1, -1, -1]])
w1 = np.matmul(sample1.T, sample1)
w1 = w1 - np.eye(w1.shape[0])
y1 = np.sign(np.matmul(sample1, w1))
w2 = w + w1
s1 = np.array([[1, 1, 1, -1], [-1, 1, -1, -1]])
y2 = np.sign(np.matmul(s1, w2))
print("output1 : \n", y2)
print("weights : \n", w2)
```

شکل 2-3

در شکل 2-4 خروجی این بخش شامل خروجی بردارهای ذخیره شده در شبکه و ماتریس وزن بروزرسانی شده را میبینیم که هر دو خروجی به درستی بردار ورودی را تداعی می کنند.

```
output1 :
 [[ 1.  1.  1. -1.]
 [-1.  1. -1. -1.]]
weights :
 [[ 0.  0.  2.  0.]
 [ 0.  0.  0. -2.]
 [ 2.  0.  0.  0.]
 [ 0. -2.  0.  0.]]
```

شکل2-4

3) در صورتیکه بردار ورودی جدید به بردارهای ورودی قبلی عمود نباشد ، شبکه قادر به ذخیره ی بردار جدید نخواهد بود بنابراین شبکه درصورتی قادر به ذخیره سازی الگوهاست که بردارهای ورودی متعامد باشند و تعداد الگوهای متعامد برابر است با تعداد بردارهای ویژه ماتریس وزن که در این مثال برابر است با 3 و ما میتوانیم حداکثر 3 بردار را ذخیره کنیم. علاوه بر دو بردار قبلی بردار [1,-1,-1,-1] را هم می توان ذخیره نمود که عمود بر دو بردار دیگر است اما هربردار دیگری را اضافه کنیم شبکه قادر به بازیابی نخواهد بود. برای نمایش این موضوع بردار sample2 = [1,1,1,1] را ذخیره می کنیم و در شکل 2-5 می بینیم که خروجی نتوانسته ورودی را تداعی کند.

```
output2 :
 [[ 1.  1.  1.  1.]
 [-1. -1. -1. -1.]
 [ 1.  1.  1.  1.]]
```

شکل 2-5

کد مربوط به این قسمت در شکل 2-6 قابل مشاهده است.

```
sample2 = np.array([[1, 1, 1, 1]])
w3 = np.matmul(sample2.T, sample2)
w3 = w3 - np.eye(w3.shape[0])
w4 = w + w1 + w3

s2 = np.array([[1, 1, 1, -1], [-1, 1, -1, -1], [1, 1, 1, 1]])
y3 = np.sign(np.matmul(s2, w4))
print("output2 : \n", y3)
```

شکل 2-6

4) این شبکه تنها قادر به ذخیره ی 3 الگوی ورودی عمود بر هم است که عبارتند از :

S1 = [1,1,1,-1]

S2 = [-1,1,-1,-1]

S3 = [1,-1,-1,-1]

در شکل 2-7 کد این قسمت را می بینیم که هر سه بردار را در شبکه ذخیره کرده ایم و بردارها دو به دو متعامدند و خروجی در شکل 2-8 آمده است که هر سه الگوی ورودی را به درستی تداعی می کند.

```
sample3 = np.array([[1, -1, -1, -1]])
w5 = np.matmul(sample3.T, sample3)
w5 = w5 - np.eye(w5.shape[0])
w6 = w + w1 + w5
s3 = np.array([[1, 1, 1, -1], [-1, 1, -1, -1], [1, -1, -1, -1]])
y4 = np.sign(np.matmul(s3, w6))
print("output3 : \n", y4)
```

شکل 2-7

```
output3 :
 [[ 1.  1.  1. -1.]
 [-1.  1. -1. -1.]
 [ 1. -1. -1. -1.]]
```

شکل2-8

سپس بردار دیگر [1,1,1,1] را به ورودی ها اضافه می کنیم و در شکل 2-9 می بینیم که شبکه قادر به بازیابی ورودی ها نیست.

```
sample4 = np.array([[1, 1, 1, 1]])
w7 = np.matmul(sample4.T, sample4)
w7 = w7 - np.eye(w7.shape[0])
w8 = w + w1 + w5 + w7
s4 = np.array([[1, 1, 1, -1], [-1, 1, -1, -1], [1, -1, -1, -1], [1, 1, 1, 1]])
y5 = np.sign(np.matmul(s4, w8))
print("output4 : \n", y5)
```

```
output4 :
 [[ 1.  1.  1.  1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [ 1.  1.  1.  1.]]
```

شکل 2-9

5) تعداد بردارهای قابل ذخیره سازی در شبکه ای که با قانون modified Hebbian آموزش دیده باشد برابر است با n-1 که n اندازه بردار ورودی است.

سوال 3– Iterative Auto-associative Net

1) بردار s=[1,1,1,-1] را مانند سوال قبل ذخیره می کنیم و کد آن در شکل 3-1 قرار داده شده است. سپس تابع وزن را براساس قانون modified hebbian محاسبه می کنیم.

```
import numpy as np

''''''''''' part1 '''''''''
s = np.array([[1, 1, 1, -1]])
w = np.matmul(s.T, s)
w = w - np.eye(w.shape[0])
```

شکل 3-1

2) در این قسمت اطلاعات 3تا از 4 ورودی را از بین میبریم و مقدار آن را صفر می گذاریم . 4 جایگشت مختلف داریم که هر 4 حالت را به عنوان ورودی به شبکه می دهیم و می بینیم که خروجی بعد از 2 تکرار قابل بازیابی است. کد این قسمت و نتیجه در شکل های 3-2 و3-3 آمده است.

```
s1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, -1]])
y1 = np.sign(np.matmul(s1, w))
print("iteration1: \n", y1)
y2 = np.sign(np.matmul(y1, w))
print("iteration2: \n", y2)
```

شکل 3-2

```
iteration1:
 [[ 0.  1.  1. -1.]
 [ 1.  0.  1. -1.]
 [ 1.  1.  0. -1.]
 [ 1.  1.  1.  0.]]
iteration2:
 [[ 1.  1.  1. -1.]
 [ 1.  1.  1. -1.]
 [ 1.  1.  1. -1.]
 [ 1.  1.  1. -1.]]
```

شکل 3-3

3) در این قسمت اطلاعات 3 تا از 4 ورودی اشتباه شده است که 4 جایگشت مختلف داریم و هر 4 تا حالت را بعنوان ورودی به شبکه می دهیم و میبینیم که قادر به بازیابی الگوهای ورودی نخواهد بود. در دو تکرار خروجی را بدست آوردیم و هر دو مقدار خروجی با هم برابر است بنابراین با تکرار بیشتر به همان نتیجه ی قبل می رسیم و شبکه هیچگاه قادر نیست الگوهای ورودی را بازیابی کند.

کد این قسمت و نتیجه در شکل های 4-3 و 5-3 آمده است.

```
s2 = np.array([[1, -1, -1, 1], [-1, 1, -1, 1], [-1, -1, 1, 1], [-1, -1, -1, -1]])
y3 = np.sign(np.matmul(s2, w))
print("iteration1: \n", y3)
y4 = np.sign(np.matmul(y3, w))
print("iteration2: \n", y4)
```

شکل 4-3

```
iteration1:
 [[-1. -1. -1.  1.]
 [-1. -1. -1.  1.]
 [-1. -1. -1.  1.]
 [-1. -1. -1.  1.]]
iteration2:
 [[-1. -1. -1.  1.]
 [-1. -1. -1.  1.]
 [-1. -1. -1.  1.]
 [-1. -1. -1.  1.]]
```

شکل 5-3

4) در شبکه ی هاپفیلد مشابه قسمت های قبل ابتدا بردار s = [1,1,1,-1] را ذخیره می کنیم. در گام بعدی 4 حالت مختلف ورودی که 3 تا از داده ها از بین رفته باشند را به شبکه می دهیم و مشاهده میکنیم شبکه قادر به بازیابی الگوی ورودی است . در گام بعدی 4 حالتی را که 3تا از داده های ورودی اشتباه شده باشند به شبکه می دهیم و شبکه قادر به بازیابی ورودی نخواهد بود و در حالتی قرار می گیرد که مقدار خروجی تغییری نخواهد کرد. کد مربوط به این قسمت در شکل 3-6 و نتایج حاصل از از دست رفتن ورودی و اشتباه شدن ورودی در شکل 3-7 آمده است.

```
''''''''''' miss data '''''
s41 = np.array([[1, 0, 0, 0]])
s42 = np.array([[0, 1, 0, 0]])
s43 = np.array([[0, 0, 1, 0]])
s44 = np.array([[0, 0, 0, -1]])
''''''''''' mistake data '''''
s45 = np.array([[1, -1, -1, 1]])
s46 = np.array([[-1, 1, -1, 1]])
s47 = np.array([[-1, -1, 1, 1]])
s48 = np.array([[-1, -1, -1, -1]])


def hop_field(s, w):
    order = np.arange(4)
    np.random.shuffle(order)
    print('orders', order)
    y = s
    wt = w.T
    for i in range(4):
        y_in = s[0][order[i]] + np.matmul(y, wt[order[i]])
        y_in = np.sign(y_in)
        s[0][order[i]] = y_in
    print(s, "\n")


hop_field(s41, w)
hop_field(s42, w)
hop_field(s43, w)
hop_field(s44, w)
```

شکل 3-6

```
output of missing :

orders [2 3 1 0]
[[ 1  1  1 -1]]

orders [2 0 1 3]
[[ 1  1  1 -1]]

orders [2 1 0 3]
[[ 1  1  1 -1]]

orders [3 2 0 1]
[[ 1  1  1 -1]]

output of mistake :

orders [1 3 0 2]
[[-1 -1 -1  1]]

orders [0 3 2 1]
[[-1 -1 -1  1]]

orders [2 1 3 0]
[[-1 -1 -1  1]]

orders [0 3 2 1]
[[-1 -1 -1  1]]
```

شکل 7-3

5) هاپفیلد تعداد الگوهای قابل ذخیره سازی در شبکه را بصورت رابطه زیر تخمین زد که در این رابطه n ابعاد بردار ورودی یا همان تعداد نورون هاست :

$$P \approx 0.15 * n$$

همچنین رابطه ی دیگری برای محاسبه تعداد الگوهای قابل ذخیره سازی وجود دارد که عبارتست از :

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

6) در حالتیکه اطلاعات شبکه از بین رفته باشد شبکه هاپفیلد در تکرار اول قادر به بازیابی ورودی است اما شبکه خودانجمنی بعد از دو تکرار اینکار را انجام می دهد بنابراین محاسبات بیشتری انجام می دهد.

و در این مورد شبکه هاپفیلد بهتر عمل می کند اما در صورتیکه اطلاعات ورودی دچار اشتباه شده باشند عملکرد هر دو شبکه مشابه هم است.

سوال 4 – Recurrent Hetero-Associative Network

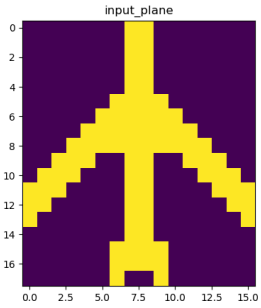
1) در این سوال بردارهای ورودی برای تصویرها (16*18) و برای متن ها (35*8) است.

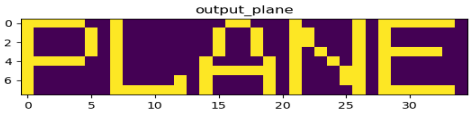
ابتدا بردار وزن به ابعاد (280*288) را محاسبه می کنیم:

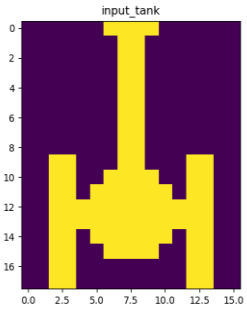
```
w = np.zeros([18 * 16, 8 * 35])
pic = np.vstack([plane_pic, tank_pic])
print(np.shape(pic))
txt = np.vstack([plane_txt, tank_txt])
print(np.shape(txt))
w = np.matmul(pic.T, txt)
print(np.shape(w))
```

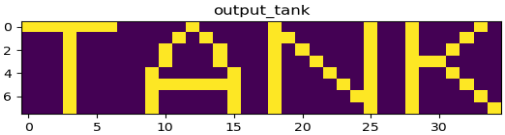
سپس با استفاده از بردار وزن ساخته شده و ورودی pic که از کنار هم قرار دادن تصویر هواپیما و تانک ایجاد شده ، سعی می کنیم خروجی را بازیابی کنیم و انتظار داریم که خروجی تولید شده با متن هواپیما و تانک یکسان باشد.

```
''''''''''''' input -> pic output -> txt'''''''''
y1 = np.sign(np.matmul(pic, w))
print(np.shape(y1))
plt.imshow(plane_pic.reshape(18, 16))
plt.title("input_plane")
plt.show()
plt.title("output_plane")
plt.imshow(y1[0].reshape(8, 35))
plt.show()
plt.imshow(tank_pic.reshape(18, 16))
plt.title("input_tank")
plt.show()
plt.title("output_tank")
plt.imshow(y1[1].reshape(8, 35))
plt.show()
```

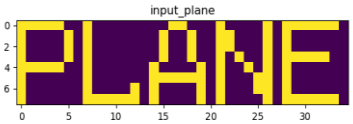


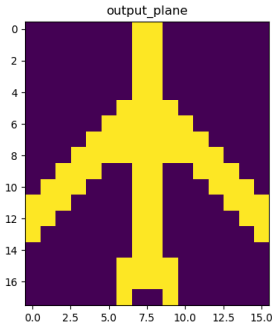


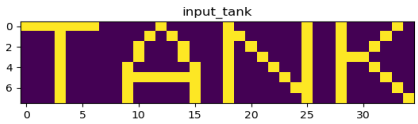


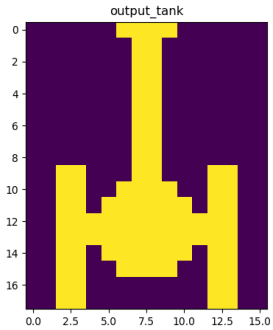


در ادامه با استفاده از بردار وزن ساخته شده و ورودی txt که از کنار هم قرار دادن متن هواپیما و تانک ایجاد شده ، سعی می کنیم خروجی را بازیابی کنیم و انتظار داریم که خروجی تولید شده با تصویر هواپیما و تانک یکسان باشد.

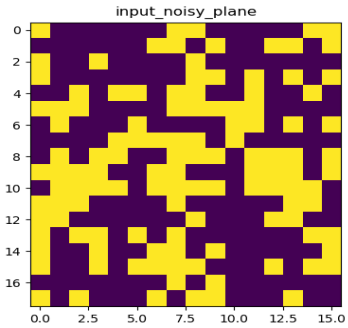


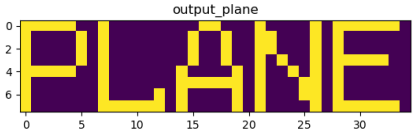


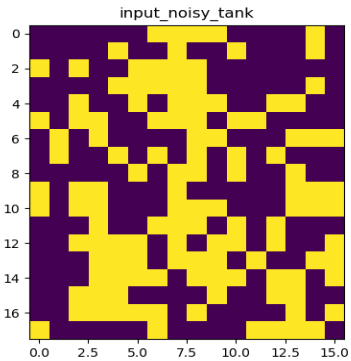


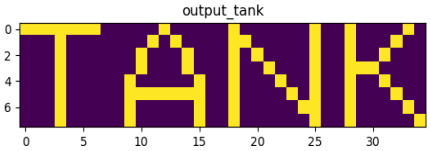


2) برای وارد کردن نویز 30 درصدی می توانیم از همان تابع mistake که در سوال اول تعریف کردیم استفاده کنیم و سپس ورودی تصویر نویزی را به شبکه می دهیم و متن را در خروجی دریافت می کنیم.









با وارد کردن 30 درصد نویز به تصویرها شبکه قادر است متن را مجددا بازیابی کند و نسبت به این میزان نویز مقاوم است.