|  |  |
| --- | --- |
| نام و نام خانوادگی | حمید رضا کاشانی |
| شماره دانشجویی | 810100441 |
| تاریخ ارسال گزارش |  |

|  |  |  |
| --- | --- | --- |
|  | به نام خدا |  |
| **دانشگاه تهران**  **دانشکده‌ مهندسی برق و کامپیوتر**  **درس رمز ارز**  **تمرین سری دوم** | | |

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

[سوال 1 1](#_Toc102255949)

[سوال ۲ 5](#_Toc102255950)

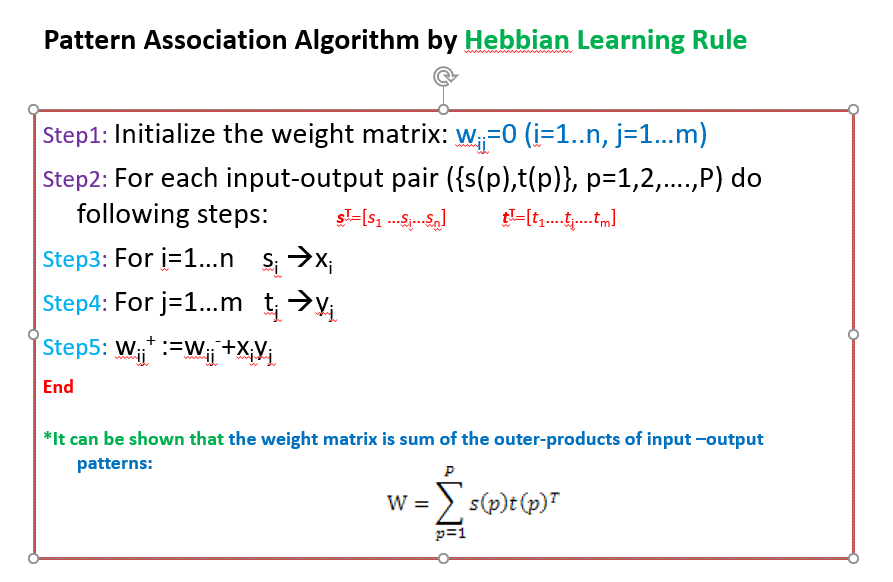
[سوال 3 6](#_Toc102255951)

[سوال 4 7](#_Toc102255952)

[سوال 5 8](#_Toc102255953)

# سوال 1

1. الگوریتم هب به طوری است که هر ورودی که باید حفظ شود رو در یه بعد قرار می دهد با یه مقدار ویژه ، فرض میکند که ورودی هایی که باید بر هم عمود باشند تا الگوریتم به خوبی کار کند ولی در صورت عمود نبودن هم با دقت خوبی الگوریتم کار می کند و هر چه تعداد ورودی ها برای حفظ کردن کمتر باشد الگوریتم بهتر کار می کند این الگوریتم به صورتی است وزن ها برابر حاصل جمع ضرب خروججی در ورودی آن وزن است و با یک بار اجرا شبکه آموزش داده می شود . این شبکه در مقابل نویز ورودی هم تا حدودی مقاوم است.



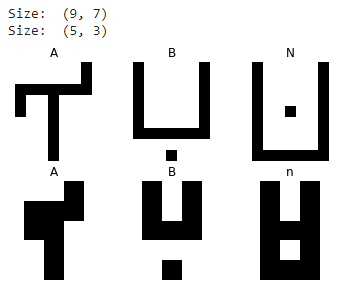
1. حال تابع های مورد نظر را برای تولید ماتریس شبکه می نویسیم.

ابتدا ورودی و خروجی ها را کنار هم می گذاریم:

S = np.hstack((A.reshape(-1, 1), B.reshape(-1, 1), N.reshape(-1, 1)))

T = np.hstack((a.reshape(-1, 1), b.reshape(-1, 1), n.reshape(-1, 1)))

ورودی و خروجی ها رو هم زیر هم رسم می کنیم:



سپس کلاس شبکه PAN با الگوریتم هب رو می نویسیم:

class PAN:

    def \_\_init\_\_(self, S, T):

        self.S = S

        self.T = T

        self.W = np.zeros((S.shape[0], T.shape[0]))

    def heb\_train(self):

        S = self.S

        T = self.T

        for i in range(T.shape[1]):

            s = S[:, i].reshape((S.shape[0], 1))

            t = T[:, i].reshape((T.shape[0], 1))

            self.W += s @ t.T

        return self.W

    def f (self, X):

        for i in range(X.shape[0]):

            for j in range(X.shape[1]):

                X[i][j] = 1 if X[i][j] >= 0 else -1

        return X

    def out\_put(self, sample):

        X = sample.reshape(-1, 1)

        return self.func(X.T @ self.W)

وزن ها و سایز وزن ها رو بعد یادگیری نمایش می دهیم

model = PAN(S,T)

w = model.heb\_train()

print(w)

w.size ,w.shape

خروجی:

[[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

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

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

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

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

[-1. -1. 1. 1. 1. 1. 1. 1. -1. -3. 1. -3. -3. 1. -3.]

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

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

[ 3. -1. 1. 1. -3. 1. 1. 1. 3. 1. -3. 1. 1. 1. 1.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-3. 1. -1. -1. 3. -1. -1. -1. -3. -1. 3. -1. -1. -1. -1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[-1. 3. -3. -3. 1. -3. -3. -3. -1. 1. 1. 1. 1. -3. 1.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. -3. 3. 3. -1. 3. 3. 3. 1. -1. -1. -1. -1. 3. -1.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]

[ 1. 1. -1. -1. -1. -1. -1. -1. 1. 3. -1. 3. 3. -1. 3.]]

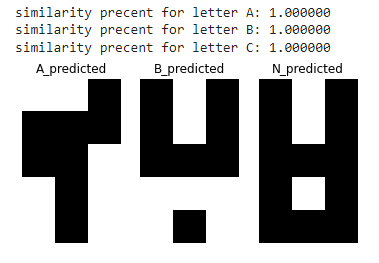
(945, (63, 15))

1. حال شبکه را تست می کنیم:

تابع تست رو می نوسیم اگر خروجی یک یدهد یعنی کاملا با مقدار اصلی خروجی برابر است:

def test\_PAN (predA, A):

    return np.sum(predA.reshape(A.shape) == A).astype('int') / 15

خروجی برای هر سه ورودی برابر است با:

به طور کامل ورودی ها با آن که باید باشد یکی است.

1. ابتدا تابعی رو برای ایجاد نویز در ورودی های درست می کنیم. سپس آخر برای 100 بار نویز دادن خروجی را تست میکنیم و در آخر درصد درست تشخیص دادن خروجی را می آوریم:

def Noise\_Net(input, percent):

    noisyOutput = input.copy()

    choosenIndices = np.random.choice(noisyOutput.size, int(percent \* noisyOutput.size / 100), replace=False)

    for index in choosenIndices:

        if noisyOutput[index] == 1:

            noisyOutput[index] = -1

        else:

            noisyOutput[index] = 1

    return noisyOutput

با 100 بار امتحان حاصل برابر:

khoroji sahih baray A ba 20 darsad noise 0.960000 darsad ast

khoroji sahih baray B ba 20 darsad noise 1.000000 darsad ast

khoroji sahih baray N ba 20 darsad noise 0.830000 darsad ast

khoroji sahih baray A ba 60 darsad noise 0.000000 darsad ast

khoroji sahih baray B ba 60 darsad noise 0.000000 darsad ast

khoroji sahih baray N ba 60 darsad noise 0.000000 darsad ast

1. همین کار را با از بین بردن اطلاعات انجام می دهیم:

تابع از بین بردن اطلاعات

def loss\_Net(input, percent):

    lossyOutput = input.copy()

    choosenIndices = np.random.choice(lossyOutput.size, int(percent \* lossyOutput.size / 100), replace=False)

    for index in choosenIndices:

        lossyOutput[index] = 0

    return lossyOutput

خروجی برای 100 بار خروجی گرفتن با از بین بردن اطلاعات:

khoroji sahih baray A ba 20 darsad noise 1.000000 darsad ast

khoroji sahih baray B ba 20 darsad noise 1.000000 darsad ast

khoroji sahih baray N ba 20 darsad noise 0.980000 darsad ast

khoroji sahih baray A ba 60 darsad noise 0.980000 darsad ast

khoroji sahih baray B ba 60 darsad noise 1.000000 darsad ast

khoroji sahih baray N ba 60 darsad noise 0.880000 darsad ast

1. مشاهده میشود که شبکه در مقابل از دست رفتن اطلاعات بسیار مقاوم تر است نسبت به نویز و تغییر علامت پیکسل و هر چه خروجی تعداد کمتری داشته باشد ابعاد آزاد و پوچ زیاد وجود دارد و این ابعاد پوچ برای از بین بردن نویز و مقاومت در مقابل نویز کارا هستند یعنی هر چه تعداد خروجی کمتر قدرت شبه در حذف اعوجاج و نویز و از بین رفتن اطلاعات بیشتر می شود.

# سوال ۲

برای تبدیل عکس ها به آرایه های بایپولار ابتدا عکس ها را آورده تبدیل به آرایه میکنیم معکوس میکنیم اعداد بزرگتر از 0 را به یک تبدیل میکنیم تا باینری شود سپس نوع داده را به int32 تبدیل می کنیم تا بتوانیم اعداد منفی هم بدهیم سپس صفر ها را به 1- تبدیل می کنیم:

from PIL import Image

import numpy as np

np\_img=[]

for i in range(3):

  img = Image.open('/content/photo/Image\_%i.png'%(i+1)).convert('L')

  np\_img += [np.array(img)]

  np\_img[i]= ~np\_img[i]  # invert B&W

  np\_img[i][np\_img[i] > 0] = 1

img1=np\_img[0]

img2=np\_img[1]

img3=np\_img[2]

img1 = img1.astype('int32')

img2 = img2.astype('int32')

img3 = img3.astype('int32')

img1[img1==0] = -1

img2[img2==0] = -1

img3[img3==0] = -1

حال برای استاده از کتابخانه های سوال قبل ورودی ها را به شکل ستون های کنار هم در می آوریم:

S = np.hstack((img1.reshape(1, 1), img2.reshape(1, 1), img3.reshape(-1, 1)))

سپس کلاس Auto-associative Net را پیاده سازی می کنیم.

class auto\_associative\_net:

    def \_\_init\_\_(self, S):

        self.S = S

        self.W = np.zeros((S.shape[0], S.shape[0]))

    def heb\_train(self):

        S = self.S

        for i in range(S.shape[1]):

            s = S[:, i].reshape((S.shape[0], 1))

            self.W += s @ s.T

        return self.W

    def heb\_train\_modified(self):

        S = self.S

        for i in range(S.shape[1]):

            s = S[:, i].reshape((S.shape[0], 1))

            self.W += s @ s.T - np.eye(S.shape[0])

        return self.W

    def f (self, X):

        for i in range(X.shape[0]):

            for j in range(X.shape[1]):

                X[i][j] = 1 if X[i][j] >= 0 else -1

        return X

    def out\_put(self, sample):

        X = sample.reshape(-1, 1)

        return self.f(X.T @ self.W)

1. حال برای ورودی های داده شده شبکه را تشکیل داده و هر دو نوع یادگیری را برای آن آنجام می دهیم:

model = auto\_associative\_net(S)

print(model.heb\_train())

model1 = auto\_associative\_net(S)

print(model1.heb\_train\_modified())

که حاصل آن که ضرایب آموزش داده شده اند به صورت زیر اند:

[[ 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.]]

[[ 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.]]

1. این الگوریتم این سه عکس را حفظ می کند و علاوه بر حفظ قابلیت تشخیص و حذف نویز ها واعوجاج ها را نیز دارد به این صورت است که ویژگی های هر عکس را در راستا برداری قرار می دعد که ایم بردار ها با هم عمود اند حال اگر اعوجاج و نویزی در ورودی باشد به علت هم راستا نبودن با بردار ورودی حذف می شود و عکس بدون اعوجاج در خروجی نمایش داده می شود .

برای هر دو مدل درستشده نتیجه ورودی بدون اعواج کاملا صحیح است:

simA = test\_NET(model.out\_put(img1), img1)

simB = test\_NET(model.out\_put(img2), img2)

simC = test\_NET(model.out\_put(img3), img3)

print('similarity precent for letter A: %f'%simA)

print('similarity precent for letter B: %f'%simB)

print('similarity precent for letter C: %f'%simC)

plt.subplot(131)

plt.imshow(model.out\_put(img1).reshape(img1.shape), cmap='binary')

plt.axis('off')

plt.title('0\_predicted')

plt.subplot(132)

plt.imshow(model.out\_put(img2).reshape(img2.shape), cmap='binary')

plt.axis('off')

plt.title('1\_predicted')

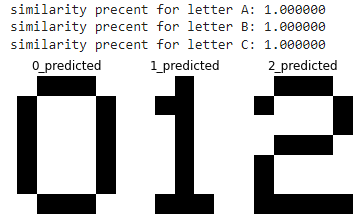
plt.subplot(133)

plt.imshow(model.out\_put(img3).reshape(img3.shape), cmap='binary')

plt.axis('off')

plt.title('2\_predicted')

plt.show()



این الگوریتم برای ورودی بدون اعوجاج کاملا درست کار میکند

3)تابع ایجاد نویز همان تابع سوال قبل است و ما هم برای هر دو مدل این تست ایجاد نویز را برای 100 سمپل انجام می دهیم:

کد آن به صورت زیر است:

def Noise\_Net(input, percent):

    noisyOutput = input.copy()

    choosenIndices = np.random.choice(noisyOutput.size, int(percent \* noisyOutput.size / 100), replace=False)

    for index in choosenIndices:

        if noisyOutput[index] == 1:

            noisyOutput[index] = -1

        else:

            noisyOutput[index] = 1

    return noisyOutput

حاصل برای Hebbian Learning Rule:

khoroji sahih baray 0 ba 20 darsad noise 0.940000 darsad ast

khoroji sahih baray 1 ba 20 darsad noise 1.000000 darsad ast

khoroji sahih baray 2 ba 20 darsad noise 0.920000 darsad ast

khoroji sahih baray 0 ba 80 darsad noise 0.000000 darsad ast

khoroji sahih baray 1 ba 80 darsad noise 0.000000 darsad ast

khoroji sahih baray 2 ba 80 darsad noise 0.000000 darsad ast

شبکه در مقابل 80 درصد نویز دوام نمی آورد

ولی برای 20 درصد نویز شبکه برای عدد 1 خوب کار می کند ولی برای عدد 0 و 2 خیر زیرا پیکسل بیشتری برای شناسایی آن مورد نیاز است

برای 0 و 2 تقریبا شبیه هم عمل می کند ولی مقداری برای 0 بهتر تشخیص می دهد

پس همه ی اعداد به یک میزان به نویز حساس نیستند اعدادی که برای شناسایی آنها به تعداد بیشتری بیت نیاز است در مقایل نویز حساس ترند

حاصل برای :Modified Hebbian Learning Rule

khoroji sahih baray 0 ba 20 darsad noise 0.840000 darsad ast

khoroji sahih baray 1 ba 20 darsad noise 0.960000 darsad ast

khoroji sahih baray 2 ba 20 darsad noise 0.910000 darsad ast

khoroji sahih baray 0 ba 80 darsad noise 0.000000 darsad ast

khoroji sahih baray 1 ba 80 darsad noise 0.000000 darsad ast

khoroji sahih baray 2 ba 80 darsad noise 0.000000 darsad ast

درالگوریتم modified شبکه برای 2 بهتر از 0 کار می کند

1. حال به جای تابع نویز از تابع loss استفاده می کنیم همان تایع سوال قبل

نتیجه برای Hebbian Learning Rule:

khoroji sahih baray 0 ba 20 darsad loss 1.000000 darsad ast

khoroji sahih baray 1 ba 20 darsad loss 1.000000 darsad ast

khoroji sahih baray 2 ba 20 darsad loss 1.000000 darsad ast

khoroji sahih baray 0 ba 80 darsad loss 0.810000 darsad ast

khoroji sahih baray 1 ba 80 darsad loss 0.950000 darsad ast

khoroji sahih baray 2 ba 80 darsad loss 0.760000 darsad ast

20 درصد از دست دادن همه قایل تشخیص اند ولی برای 80 درصد به ترتیب 1 بهترین و بعد 0 بهتر از 2 است اما فرق چندانی ندارن

نتیجه برای Modified Hebbian Learning Rule :

khoroji sahih baray 0 ba 80 darsad loss 1.000000 darsad ast

khoroji sahih baray 1 ba 80 darsad loss 1.000000 darsad ast

khoroji sahih baray 2 ba 80 darsad loss 1.000000 darsad ast

khoroji sahih baray 0 ba 80 darsad loss 0.470000 darsad ast

khoroji sahih baray 1 ba 80 darsad loss 0.540000 darsad ast

khoroji sahih baray 2 ba 80 darsad loss 0.490000 darsad ast

برای این مدل هم 1 بهتر از 0 و2 و همینطور 2 بهتر از 0 است

# سوال 3

1. شبیه شبکه Recurrent Auto-Associative NET است که در آن ورودی به صورت بازگشتی از خروجی به دست می آمد تا آن که شبکه به خروجی ثابتی میل کند و در آن تنها از ورودی در بار اول استفاده می شد و ورودی های بعدی شبکه خروجی های لحظه قبل آن بودند اما تفاوت آن با این شبکه Discrete Hopfield Net این است که یک هر بار تنها یکی از یونیت های شبکه به روز رسانی می شوند و تفاوت دوم این است که در هر بار بروز رسانی علاوه بر خروجی لحظه قبل ورودی اعمال شده به شبکه هم در الگوریتم بروز رسانی دخالت دارد. و هر یونیت به صورت اتفاقی از بین یونیت های بروز نشده انتخاب می شوند تا یک ایپاک تمام شود . این شبکه بازدهی بهتری نسبت به شبکه های قبلی دارد و در برابر نویز و اعوجاج مقاومت بیشتری از خود نشان می دهند.
2. عکس را آورده تبدیل باینری می کنیم و سپس سایز آن را به سایز استاندارد تغییر می دهیم

برای چندین ترشولد خروجی را می آوریم:

img = Image.open('/content/train.jpg').convert('L')

np\_img = np.array(img)

print(np\_img)

#np\_img= ~np\_img  # invert B&W

#print(np\_img)

teta=85

np\_img[np\_img<= teta] = 0

np\_img[np\_img > teta] = 1

import cv2

height = 68

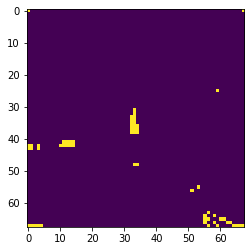
width = 68

img\_resized = cv2.resize(np\_img, (width, height))

برای ترشولد 85 :



برای ترشولد 125:

برای ترشولد 185:

برای ترشولد 100 :



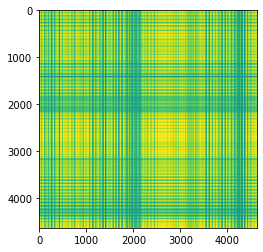
به نظر ترشولد 85 برای این مدل وضوح بیشتری دارد.

1. ماتریس وزن ها از ضرب داخلی ورودی در خودش منهای ماتریس همانی حاصل می شود برای درک بهتر عکس ماتریس وزن ها را چاپ می شود

w =S.reshape(-1, 1) @ S.reshape(1, -1) - np.eye(S.size)

plt.imshow(w)

w



1. الگوریتم bipolar داده شده در اسلاید ها را پیاده سازی می کنیم والگوریتم حوری است که نود بعدی برای یادگیری به صورت رندوم انتخاب می شود و خروجی را در هر 50 بار بروز رسانی می کشیم:

کد مربوط:

x = S.reshape(1, -1)

n=0

nn=0

y = np.copy(x)

out = None

def comparator( input, threshold):

  if input > threshold:

    return 1

  elif input < threshold:

    return -1

  else:

    return input

randiter = np.random.choice(np.arange(64\*64), 64\*64, replace=False)

axes=[]

fig=plt.figure(figsize=(64, 64))

for i in randiter:

  y\_in = x[0][i] + np.sum(y[0] \* w[:, i])

  y[0][i] = comparator(y\_in, 0)

  out = np.copy(y)

  n=n+1

  if n%50 == 0 :

    nn =nn+1

    axes.append( fig.add\_subplot(11, 8, nn) )

    subplot\_title=("Subplot"+str(nn))

    axes[-1].set\_title(subplot\_title)

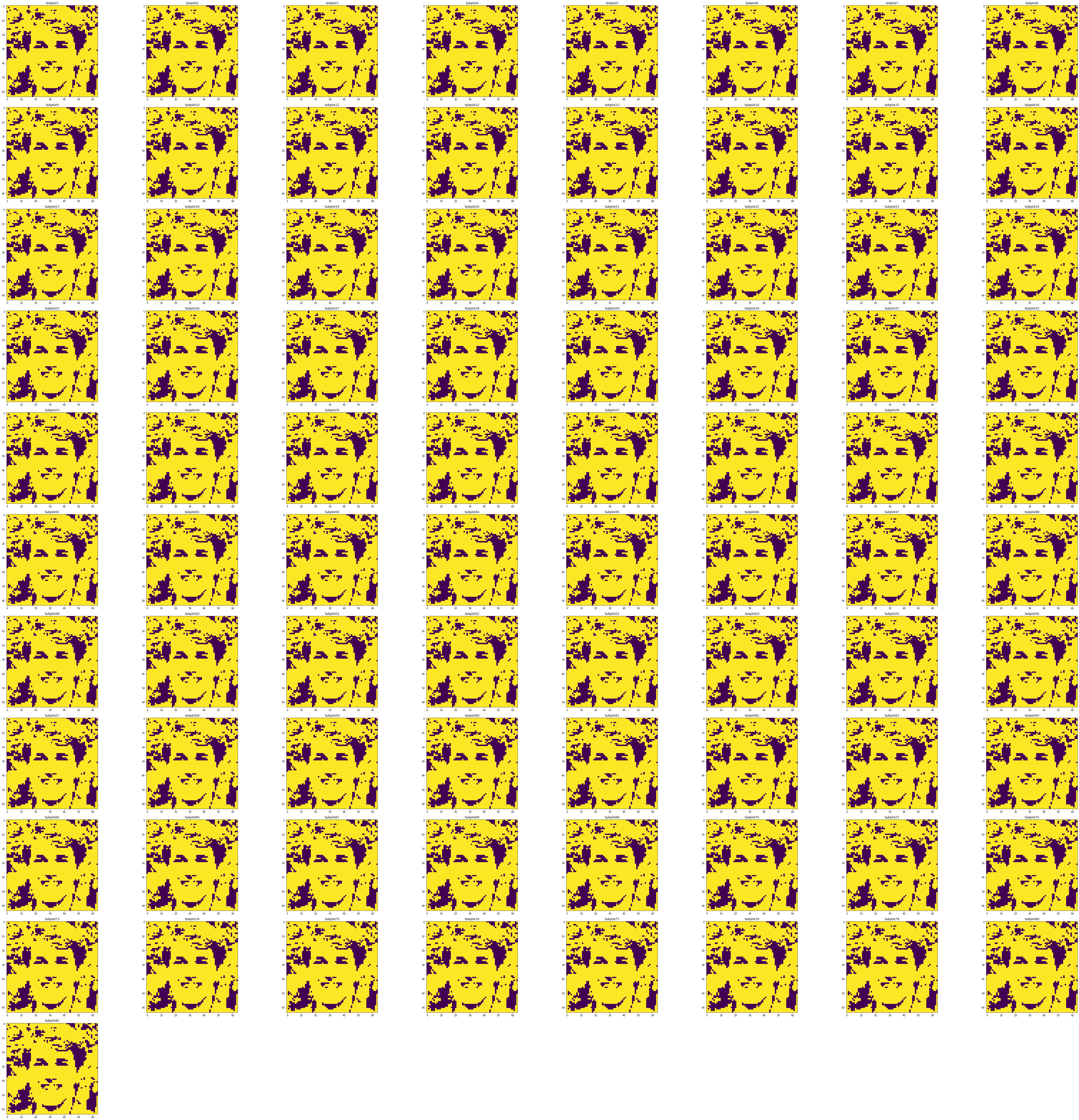
    plt.imshow(out.reshape(64,64))

    #print(out.reshape(64,64))

fig.tight\_layout()

plt.show()

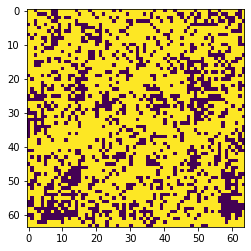
خروجی به صورت زیر است:



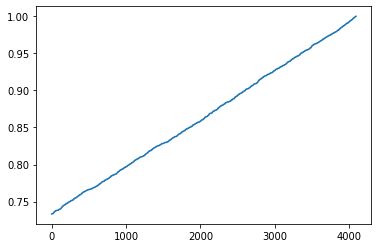
همان طور که مشاهده می شود خروجی از همان تکرار های اولیه به صورت صحیح کار می کند مطابق انتظار.

1. حال یک تابع مقایسه گر می سازیم تا نمودار آن را بر حسب تکرار ها رسم کنیم برای عر کدام از داده های نویزی تست.

عکس تست یک:



نتایج شبکه:

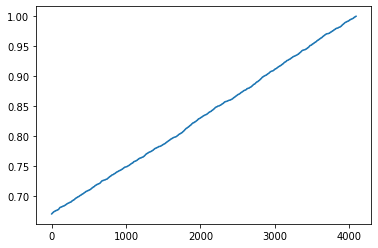




عکس تست 2:

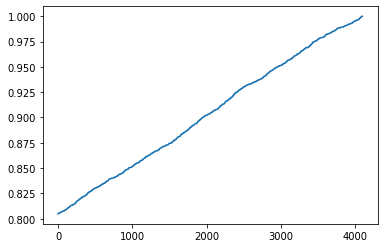


نتایج:





برای تست سوم:





# سوال 4

1. الگوریتم به طوری است که هر بلوکی که زود تر به ماینر ها برسد ماینر ها در صورت مساوی بودن زنجیره آن را برای ادامه کار انتخاب می کنند حال که به طور همزمان بلوک بعدی توسط دو ماینر پیدا شده است هر کدام از ماینر ها به علت تاخیر نود های کناری خود را به ادامه دادن زنجیره آنها وا میدارد و شبکه به دو شاخه تقسیم می شود هر شاخه ای که بلوک های بعدی خود را زودتر پیدا کند شاخه ی بزرگتر و اصلی میشود و بلوک های شاخه ی دیگر از بین میرود شاخه ای که از ابتدا ماینر بیشتری دارد شانسش برای شاخه ی اصلی شدن بیشتر است در واقع با احتمال خوبی هر ماینری که بلوک استخراج شده ی خود را با تاخیر کمتری به تعداد بیشتری برساند شاخه ی اصلی با آن می رسد و دیگری یتیم می شود.
2. تاخیر شبکه و معماری آن و همینطور قدرت پردازی شاخه ی در رقابت برای یتیم نشدن و همینطور حاصل ضرب دلتا در لاندا پارامتر هایی هستند که رو ی نرخ یتیم شدن تاثیری دارند و می شود با انجام تحلیل های آماری همانند مارکف چین ساختن فرمولی برای نرخ یتیم شدن پیدا کرد .(مقاله استفاده از تکنیک های آماری و مارکف چسن برای پیدا کردن نرخ یتیم شده موجود است)
3. به علت تغییر زیاد توپولوژی و ساختار شبکه و تغییر تاخیر های بین نود ها نرخ یتیم شدن برای بلوک ها مختلف متفاوت می شود و به حاصل ضرب دلتا در لاندا بستگی داردکه هر چه بیشتر شود احتمال یتیم شدن بالا تر است اما ما به صورت تجربی می توانیم نسبت تعداد کل زنجیره را در مقطع زمانی خواص با تعداد بلوک های یتیم شده در همان زمان را محاسبه کنیم تا احتمال یتیم شدن به دست بیاید و به طور کلی باید بلوک های یتیم هم فرایند پوآسن باشند و زمان بینشون توزیع نمایی داشته باشد پس میانگین و واریانس آن را در یک بازه زمانی می توان به طور تجربی محاسبه کرد .
4. خیر ولی احتمالش برای پیروزی کمتر می شود و برای رقابت معمولا باید بلوک های خالی درست کند تا شاید بتواند جلوی یتیم شدن خود را بگیرد با پایین آوردن تاخیرش برای بلوک های بعدی.
5. تفاوت چندانی نباید داشته باشد ولی به علت توپولوژی شبکه امکان دارد برای بعضی ماینر ها تاخیر بیشتری نسبت به بقبه وجود داشته باشد که خود آن به معنی نرخ یتیم شدن بالا تر است ولی ببرای اکثر نود ها زیاد متفاوت نیست این نرخ اما به طور کلی بایستی متفاوت باشد.

**سوال 5**

1. در صورتی که سرویس دهنده توان پردازشی بالایی نداشته باشد امکان حمله دوبار خرج برای آن وجود ندارد زیرا در این حمله باید زنجیره را به دو قسمت تقسیم کرد و در هر شاخه پول کلاینت مورد نظر را خرج کرد در صورتی که سرویس دهنده بتواند این دو شاخه را بیشتر از 6 بلوک در تعادل نگه دارد و یا شاخه ی اصلی را به طور کلی از بلوک هی قبل تر جدا کند و زنجیره کلی را در ادامه زنجیره خود کند توانسته حمله دابل اسپند بزند ولی این حمله به بیش از 50 درصد توان کل شبکه نیاز دارد که صرفه اقتصادی این کار را زیر سوال می برد دو حمله ی دیگر مربوط به دابل اسپند وجود دارد که حمله فینی و رقابتی نام دارد که هر دو اگر کلاینت به اندازه 6 بلوک بایستد و سریع تراکنش را تایید نکن قابل اجرا نیست به طور کلی بیت کوین جوری طراحی شده است که امنیتش را با وجود آگاهی همه از پیام هایش حفظ شود با کمک کلید عمومی و خصوصی و هش کردن. پس مشاهده ارتباطات کلاینت با شبکه تفاوت زیادی برای ایجاد حمله ندارد و فقط سرویس دهنده می تواند پیام کلاینت را دیر تر از زمان مورد انتظار بفرستد و تاخیرش را اضافه کند که این کار شاید در حمله فینی موثر باشد ولی این حمله با زود تایید نکردن تراکنش توسط کلاینت بر طرف می شود.