



به نام خدا

دانشگاه تهران
دانشکده مهندسی برق و کامپیوتر
درس شبکه‌های عصبی و یادگیری عمیق
تمرین سری سوم

نام و نام خانوادگی	محسن فیاض
شماره دانشجویی	810100524
تاریخ ارسال گزارش	23 اردیبهشت 1400

فهرست گزارش سوالات

سوال 1 – Pattern Association using Hebbian Learning Rule

- 1 (1) الگوریتم Hebbian Learning Rule را توضیح دهید.
- 2 (2) شبکه ای طراحی کنید که با گرفتن ورودی 9×7 خروجی 5×3 را تولید کند. وزن های شبکه را به کمک Hebbian Learning Rule آپدیت کنید و مقدار آن را به صورت یک ماتریس نشان دهید
- 3 (3) آیا شبکه شما توانسته خروجی مطلوب هر ورودی را تولید کند؟ خروجی هر ورودی را رسم کنید.
- 4 (4) کوچکترین ابعادی که شبکه می تواند ورودی 7×9 را به خروجی مطلوب برساند چیست؟
- 5 (5) ورودی 7×9 را با اضافه کردن 20 و 60 درصد noise (تبدیل کردن اعداد 1 و -1 به صورت تصادفی) به شبکه برای هر دو ابعاد خروجی بخش 2 و 4 اعمال کنید. خروجی شبکه چیست ؟ در چند درصد مواقع خروجی درست تشخیص داده شد؟
- 6 (6) ورودی 7×9 را با از بین بردن 20 و 60 درصد اطلاعات (تبدیل کردن اعداد 1 و -1 به صورت تصادفی به صفر) به شبکه برای هر دو ابعاد خروجی بخش 2 و 4 اعمال کنید. خروجی شبکه چیست ؟ در چند درصد مواقع خروجی درست تشخیص داده شد؟
- 7 (7) مقاومت شبکه در برابر نویز بیشتر است یا از دست دادن اطلاعات؟ تاثیر ابعاد خروجی بر مقاومت شبکه چیست ؟

سوال 2 – Auto-associative Net

- 1 (1) وزنهای شبکه را با استفاده از دو قاعده Hebbian Learning Rule Modified Hebbian Learning Rule بدست آورده و گزارش کنید.
- 2 (2) کارایی الگوریتم Hebbian Learning Rule با اعمال تصاویر پوشه ی Images_Q2 به عنوان ورودی شبکه بررسی کنید.
- 3 (3) الف) کارایی الگوریتم Hebbian Learning Rule را بر روی تصاویر پوشه ی Images_Q2 با اعمال نویز 20% و نیز 80% به عنوان ورودی شبکه بررسی کنید. ب) آیا همه ی اعداد به یک میزان نسبت به نویز حساس هستند یا خیر؟ دلیل انتخاب بلی یا خیر خود را توضیح دهید. در صورتی که جواب شما بلی است، حساس ترین عدد نسبت به نویز کدام است؟ (برای اعمال نویز کافی است به جای 1، -1 و به جای -1، 1 قرار دهید)
- 4 (4) قسمت الف گام قبل را برای حالتی که داده ها از بین رفته باشند (به جای مقادیر 1 و -1 صفر قرار گیرد) تکرار کنید.
- 5 (5) (امتیازی) در این گام میخواهیم تعداد تصاویر را افزایش دهیم. بدین منظور به سراغ پوشه Extra بروید. اگر بررسی کنید مشاهده می فرمایید که الگوریتمهای ذکر شده در گام قبل کارایی مطلوبی را از خود نشان نمیدهند. در این گام از شما میخواهیم روش شبه معکوس را پیاده کرده و قدرت به خاطر سپاری شبکه برای تصاویر موجود در پوشه Extra را با تکرار گام 3 مورد بررسی قرار دهید. روش را به مختصر شرح دهید. (توجه نمایید به توضیح الگوریتم بدون پیاده سازی یا پیاده سازی بدون توضیح مختصر الگوریتم نمره ای تعلق نمیگیرد.)

سوال 3 – Discrete Hopfield Network

- 1 (1) در مورد Discrete Hopfield Net مختصر توضیح دهید.

2) ابتدا سائز عکس ها را به 64×64 در بیاورید و سپس تصویر حاصل را به فرم Bipolar در بیاورید. (یعنی یک threshold بگذارید و پیکسل هایی با مقدار بیشتر از threshold مقدار 1 پیکسل هایی با مقدار کمتر از threshold مقدار -1 نسبت دهید). عکس حاصل را در گزارش قرار دهید. (نکته: مقادیر threshold مختلف تست کنید. این threshold مقدار عددی است بین 0 تا 255)

3) ماتریس وزن ها را بر اساس تصویر قسمت قبلی بسازید.

4) با کمک ماتریس وزن های قسمت 3 سعی کنید تصویر اصلی را بازبایید. در هر 50 iteration عکس حاصل را چاپ کنید.

5) نمودار Hamming Distance per iteration هر تصویر تست را رسم کنید.

سوال 4 – Bidirectional Associative Memory

ماتریس وزن را بدست آورید و در گزارش مکتوب کنید

کارایی شبکه در بازیابی اطلاعات از هر دو جهت را بررسی کنید و نتایج را به طور کامل گزارش نمایید.

در این گام به صورت تصادفی ابتدا 10% بیتها و سپس 20% بیتها برای هر یک از ورودیها در هر دو جهت را نویزی کرده و درصد خروجی درست شبکه را گزارش کنید. (دقت کنید که در این گام میبایست تشابه را بر اساس تعداد بیتهای برابر خروجی شبکه و خروجی مطلوب بر حسب درصد گزارش کنید. برای آنکه نتایج شما قابل تعمیم باشد، می بایست در یک حلقه ی صدتایی این میزان را بررسی کنید و نهایتاً 6 عدد را به ازای 10% نویز تصادفی برای رفت و برگشت و نیز 6 عدد به ازای 20% نویز تصادفی برای رفت و برگشت بر حسب درصد در یک جدول ارائه کنید. همچنین برای اعمال نویز کافی است به جای 1، -1 و به جای -1، 1 قرار دهید.)

حال یک شخصیت دیگر را در کنار سه شخصیت قبلی در آموزش شرکت دهید و بررسی کنید چه تعداد از خروجیها توسط ورودیها و چه تعداد از ورودیها توسط خروجیها قابل بازیابی است؟ آیا کارایی شبکه کاهش یافته است یا خیر؟ دلیل خود را شرح دهید.

سوال 1 – Pattern Association using Hebbian Learning Rule

1) الگوریتم Hebbian Learning Rule را توضیح دهید.

قانون هب ساده ترین و رایج ترین روش برای تعیین وزن شبکه عصبی حافظه انجمنی است. این روش یک شبکه عصبی تک لایه است از ورودی به خروجی و از ساده ترین روش های یادگیری است. در این الگوریتم ابتدا وزنها برابر 0 قرار داده می شوند و سپس به ازای هر داده ورودی و خروجی s و t ، ورودی x_i به s_i و y_j به t_j مقدار دهی می شود و سپس هر وزن w_i بعلاوه ضرب x_i و y_i می شود. البته باید توجه داشت که این توضیح صرفاً برای راحتی توصیف است. در عمل و محاسبات می توان به راحتی ورودی و خروجی ها را ماتریس گرفت و ماتریس وزنها را به صورت جمع ضرب خارجی ورودی در خروجی به ازای داده ها دانست که در صفحه 104 کتاب توضیح داده شده است $W = \sum_{p=1}^P s^T(p)t(p)$.

2) شبکه ای طراحی کنید که با گرفتن ورودی 7*9 خروجی 3*5 را تولید کند. وزن های شبکه را به کمک Hebbian Learning Rule آپدیت کنید و مقدار آن را به صورت یک ماتریس نشان دهید

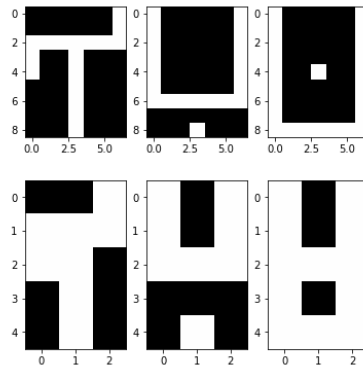
ابتدا ورودی ها و خروجی ها را به صورت مناسب بارگذاری می کنیم و آنها را نمایش می دهیم.

```
def draw(image):
    if len(image) == 63:
        image = (image.reshape(9, 7) + 1) * 127
    elif len(image) == 15:
        image = (image.reshape(5, 3) + 1) * 127
    img = Image.fromarray(image)
    plt.imshow(img)

def draw_images(arrays):
    for i in range(len(arrays)):
        plt.subplot(1, 3, i + 1)
        draw(arrays[i])
    plt.show()

inputs = np.array([input_1, input_2, input_3])
outputs = np.array([output_1, output_2, output_3])

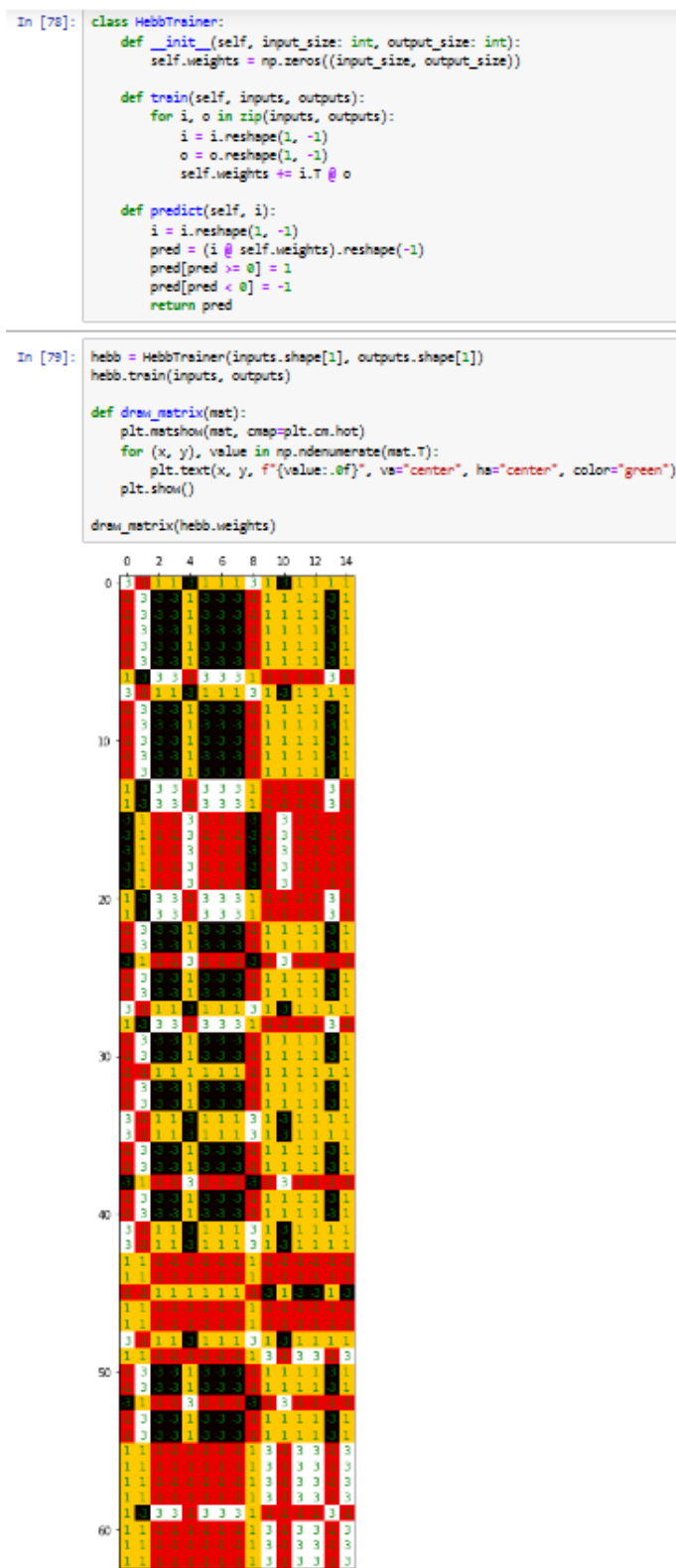
draw_images(inputs)
draw_images(outputs)
inputs.shape, outputs.shape
```



: ((3, 63), (3, 15))

شکل 1 - بارگذاری و نمایش دیتاست

سپس برای آموزش hebb کلاسی به شکل زیر پیاده سازی شد و نتیجه آموزش وزنها نمایش داده شد.



شکل 2 - آموزش هب

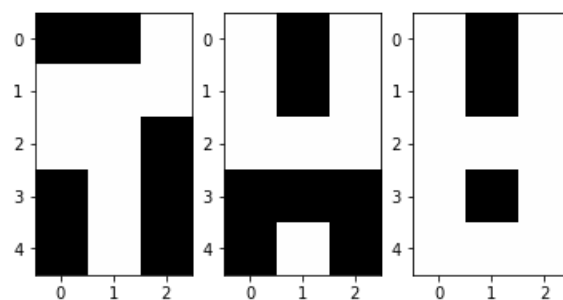
3) آیا شبکه شما توانسته خروجی مطلوب هر ورودی را تولید کند؟ خروجی هر ورودی را رسم کنید.

برای تست مدلی که آموزش دیده ورودیها را می‌دهیم تا با استفاده از تابع predict که تعریفش قبل تر آمد و صرفاً یک ضرب ماتریسی است و فعالسازی خروجی به دست آید و آن را نمایش دادیم.

```
: preds = [hebb.predict(i) for i in inputs]
draw_images(preds)

def compute_accuracy(true, preds):
    tp, all = 0, 0
    for i in range(len(preds)):
        for j in range(len(preds[i])):
            if preds[i][j] == true[i][j]:
                tp += 1
        all += 1
    return tp / all

print("Accuracy:", compute_accuracy(outputs, preds))
```



Accuracy: 1.0

شکل 3 - خروجی مدل هب

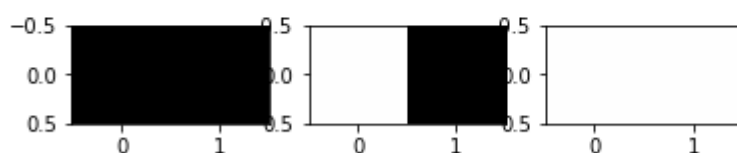
همانطور که دیده می‌شود، خروجیها دقیقاً همانند چیزی هستند که در بخشهای قبل نمایش داده شد و همچنین دقت 100 درصد است، بنابراین مدل توانسته خروجی مطلوب را تولید کند.

4) کوچکترین ابعادی که شبکه می تواند ورودی 7*9 را به خروجی مطلوب برساند چیست؟

از آنجا که سه ورودی برای به حافظه سپردن داریم، بنابراین 3 مقدار برای آن کافیست که در سیستم بایپولار که استفاده می کنیم می توان با 2 خانه حافظه به مقادیر 1,1 و 1,-1 و 1,-1 خروجی ها را داشت. نتیجه اجرا با این ابعاد خروجی در زیر دیده می شود.

```
small_hebb = HebbTrainer(inputs.shape[1], outputs_small.shape[1])
small_hebb.train(inputs, outputs_small)

preds = [small_hebb.predict(i) for i in inputs]
draw_images(preds)
print("Accuracy:", compute_accuracy(outputs_small, preds))
```



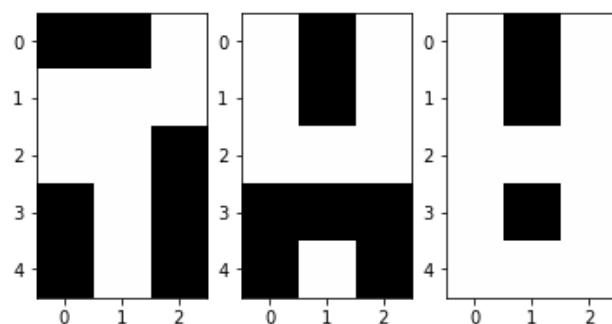
Accuracy: 1.0

شکل 4 - خروجی مدل هب برای ابعاد خروجی کوچک

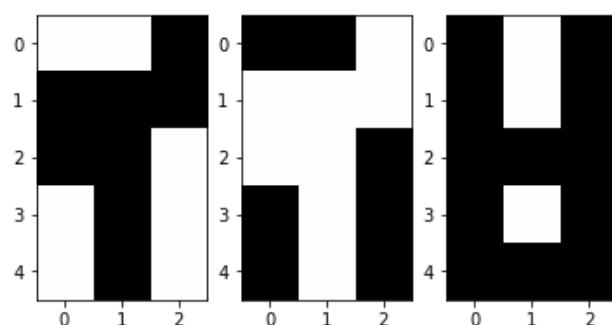
همانطور که دیده می شود مدل توانسته دقت 100 درصد را کسب کند و خروجی های دو بعدی را به درستی خروجی دهد.

5) ورودی 7*9 را با اضافه کردن 20 و 60 درصد noise (تبدیل کردن اعداد 1 و -1 به صورت تصادفی) به شبکه برای هر دو ابعاد خروجی بخش 2 و 4 اعمال کنید. خروجی شبکه چیست؟ در چند درصد مواقع خروجی درست تشخیص داده شد؟

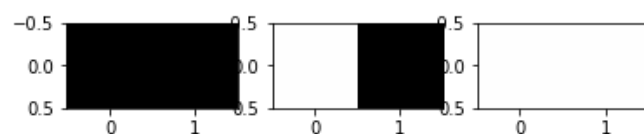
برای ایجاد نویز تابع زیر پیاده سازی شد و سپس با استفاده از آن ورودی های نویزدار ساخته و به مدل داده شد. خروجی ها نیز چاپ و دقت کلی محاسبه شد. (دقت روی تک تک پیکسل ها هست).



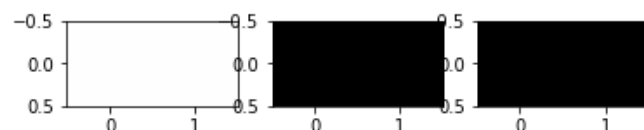
Noise 20.0% - Average Accuracy (over 1000 runs): 92.21%



Noise 60.0% - Average Accuracy (over 1000 runs): 0.92%



Noise 20.0% - Average Accuracy (over 1000 runs): 92.29%



Noise 60.0% - Average Accuracy (over 1000 runs): 4.19%

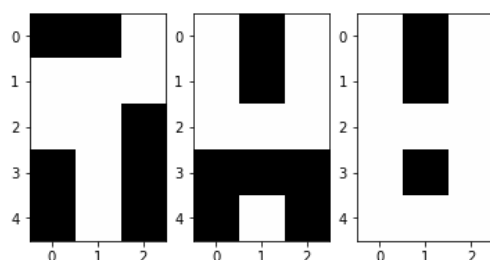
شکل 5 - نتیجه نویز

همانطور که دیده می‌شود، با نویز 20 درصد، همچنان مدل توانسته robust عمل کند و خروجی های درست را در اکثر مواقع دقیقاً بدهد و دقت میانگین 92.21% را کسب کند. اما با افزایش نویز به 60 درصد، دیگر مدل اکثراً هیچکدام از 3 مورد را نتوانسته درست تشخیص دهد و خروجی های اشتباهی داده است و میانگین دقت زیر یک درصد شده است. البته باید دقت داشت که 60 درصد نویز واقعاً می‌تواند ورودی را به کلی تغییر دهد و شاید نباید چندان انتظار هم داشت که خروجی مناسب با این مقدار نویز داده شود. مخصوصاً در مورد شبکه ساده‌ای که

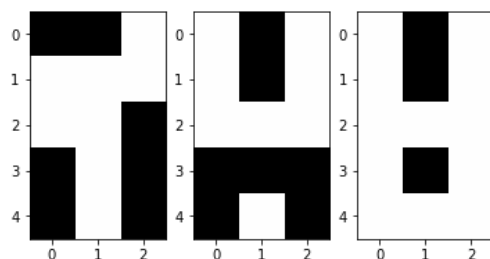
بررسی می‌کنیم. اما در هر حالتهای ابعاد خروجی کمتر، در حالت 20 درصد نویز دقت 92.29 و در نویز 60 درصد دقت 4.19 را داریم که در هر دو مورد بیشتر از ابعاد بزرگتر خروجی است.

6) ورودی 7*9، با از بین بردن 20 و 60 درصد اطلاعات (تبدیل کردن اعداد 1 و 0 به صورت تصادفی به صفر) به شبکه برای هر دو ابعاد خروجی بخش 2 و 4 اعمال کنید. خروجی شبکه چیست؟ در چند درصد مواقع خروجی درست تشخیص داده شد؟

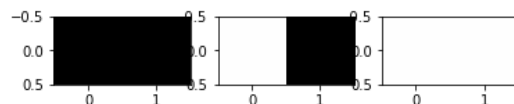
در این مورد هم ابتدا از بین بردن اطلاعات را پیاده کرده و نتیجه آن را نمایش دادیم.



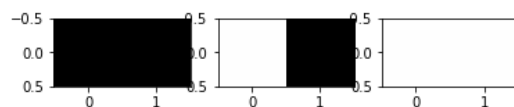
Loss 20.0% - Average Accuracy (over 100000 runs): 99.81%



Loss 60.0% - Average Accuracy (over 100000 runs): 94.12%



Loss 20.0% - Average Accuracy (over 100000 runs): 99.80%



Loss 60.0% - Average Accuracy (over 100000 runs): 94.16%

شکل 6 - نتیجه از بین بردن

در این مورد هم مدل توانسته 20 درصد از بین رفتن داده را در اکثر مواقع کاملاً تحمل کند و به دقت میانگین 99.81 و 99.80 در دو حالت خروجی برسد، اما 60 درصد زیاد بوده و باعث شده مدل به ترتیب 94.12 و 94.16 میانگین دقت داشته باشد.

(7) مقاومت شبکه در برابر نویز بیشتر است یا از دست دادن اطلاعات؟ تاثیر ابعاد خروجی بر مقاومت شبکه چیست؟

مقاومت شبکه همانطور که دیده شد در برابر از دست دادن اطلاعات بیشتر است. این موضوع را می‌توان طبق ساختار کاری که انجام می‌شود نیز توضیح داد. برای ساختن خروجی‌ها، ورودی‌ها را در ماتریس وزنی که به دست آمده است ضرب می‌کنیم. به صورتی می‌توان این کار یک تبدیل خطی دانست. به این معنا که هر مقدار ورودی تاثیری روی خروجی می‌گذارد و در مجموع ترکیب آنها خروجی را می‌سازند. حالا اگر نویز داشته باشیم که مثلاً 1 شده باشد -1، تاثیر منفی در نتیجه نهایی گذاشته می‌شود. در حالیکه از دست دادن داده و صفر شدن آن، صرفاً تاثیر آن ورودی خاص را خنثی می‌کند و باز بقیه خانه‌های ورودی شاید بتوانند آن را جبران کنند. در مورد ابعاد نیز دیدیم که هر چه ابعاد خروجی کوچکتر شد، مقاوم‌تر شد. در این مورد، ابعاد کوچکتر خروجی باعث کوچکتر شدن ابعاد ماتریس وزن‌ها نیز می‌شود. و در عمل اگر ابعاد خروجی بزرگتر باشد، احتمال خطا نیز بیشتر می‌شود چون هر کدام از مقادیر ابعاد خروجی که اشتباه داده شود کل آن خروجی از دست می‌رود. اما در ابعاد کمتر احتمال پیش آمدن خرابی نیز کمتر است چون تعداد ابعاد کوچکتر است. در اعداد هم دیدیم که ابعاد کوچکتر خروجی باعث مقاومت بیشتر شده بود.

سوال 2 – Auto-associative Net

ابتدا داده‌های داده شده را می‌خوانیم و برای اطمینان از صحت آن، نمایش می‌دهیم.

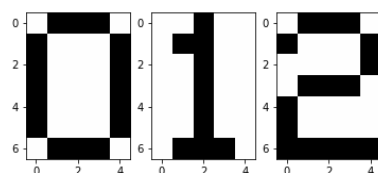
0) Load data

```
def load_image(img_path):
    img = Image.open(img_path)
    img.load()
    data = np.asarray(img, dtype="int32")
    data = np.mean(data, axis=-1) # RGB -> Grayscale
    data[data < 127] = -1
    data[data >= 127] = 1 # Bipolar
    data = data.flatten()
    return data

def draw_image(np_array):
    image = (np_array.reshape(7, 5) + 1) * 127
    img = Image.fromarray(image)
    plt.imshow(img)

def draw_images(arrays):
    for i in range(len(arrays)):
        plt.subplot(1, len(arrays), i + 1)
        draw_image(arrays[i])
    plt.show()

dataset = []
for i in range(1, 4, 1):
    dataset.append(load_image(f"Images_Q2/Image_{i}.png"))
draw_images(dataset)
```



شکل 7 - خواندن دیتا

1) وزنهای شبکه را با استفاده از دو قاعده Hebbian Learning Rule Modified و گزارش کنید.

در زیر کلاسی را طراحی کردیم که می‌تواند هب را به صورت عادی و به صورت تغییر کرده بدهد. همچنین وزنهای بعد از آموزش نمایش داده شدند.

```
class ModifiedHebbTrainer:
    def __init__(self, input_size: int, output_size: int, modified=False):
        self.weights = np.zeros((input_size, output_size))
        self.modified = modified

    def train(self, inputs, outputs):
        for i, o in zip(inputs, outputs):
            i = i.reshape(1, -1)
            o = o.reshape(1, -1)
            self.weights += i.T @ o
            # Modification
            if self.modified:
                np.fill_diagonal(self.weights, 0)

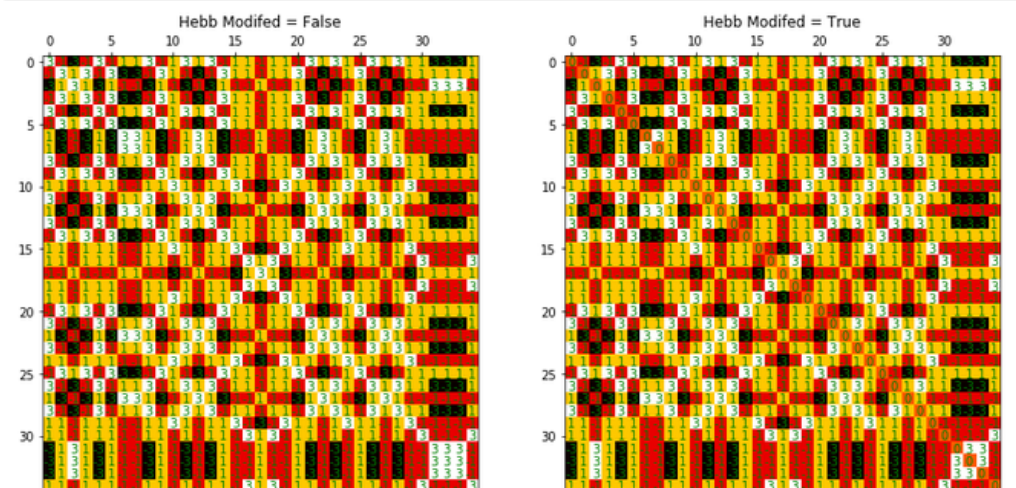
    def predict(self, i):
        i = i.reshape(1, -1)
        pred = (i @ self.weights).reshape(-1)
        pred[pred >= 0] = 1
        pred[pred < 0] = -1
        return pred

    def draw_weights(self):
        mat = self.weights
        plt.matshow(mat, cmap=plt.cm.hot, fignum=False)
        for (x, y), value in np.ndenumerate(mat.T):
            plt.text(x, y, f"{value:.0f}", va="center", ha="center", color="green")
        plt.title(f"Hebb Modified = {self.modified}")
```

```
hebb = ModifiedHebbTrainer(dataset.shape[1], dataset.shape[1], modified=False)
hebb.train(dataset, dataset)

modified_hebb = ModifiedHebbTrainer(dataset.shape[1], dataset.shape[1], modified=True)
modified_hebb.train(dataset, dataset)

plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
hebb.draw_weights()
plt.subplot(1, 2, 2)
modified_hebb.draw_weights()
```

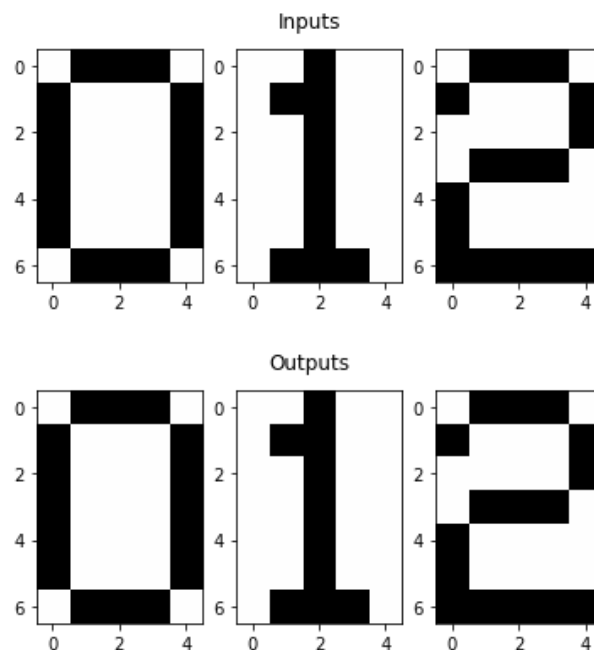


شکل 8 - آموزش و وزنهای دو حالت هب

2) کارایی الگوریتم Hebbian Learning Rule با اعمال تصاویر پوشه‌ی Images_Q2 به عنوان ورودی شبکه بررسی کنید.

در زیر اعمال مدل روی ورودی‌ها و نمایش خروجی نمایش داده شده است.

```
preds = [hebb.predict(d) for d in dataset]
draw_images(dataset, title="Inputs")
draw_images(preds, title="Outputs")
```



شکل 9 - آموزش و وزنهای دو حالت هب

همانطور که دیده می‌شود هر 3 را به راحتی توانسته بازسازی کند.

3 الف) کارایی الگوریتم Hebbian Learning Rule را بر روی تصاویر پوشه ی Images_Q2 با اعمال نویز 20% و نیز 80% به عنوان ورودی شبکه بررسی کنید.

ب) آیا همه ی اعداد به یک میزان نسبت به نویز حساس هستند یا خیر؟ دلیل انتخاب بلی یا خیر خود را توضیح دهید. در صورتی که جواب شما بلی است، حساس ترین عدد نسبت به نویز کدام است؟ (برای اعمال نویز کافی است به جای 1، -1 و به جای -1، 1 قرار دهید)

الف)

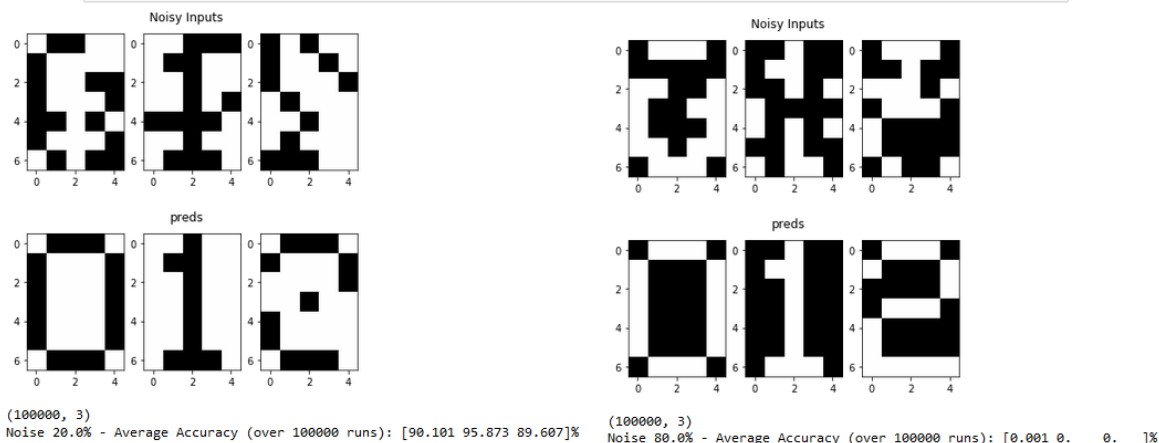
در کد پیاده سازی شده زیر ابتدا نویز اضافه می شود و سپس اینکار 1000 بار تکرار می شود و در نهایت دقت در این تکرارها به ازای هر یک از سه کلاس گزارش می شود.

```
In [103]: def add_noise(array, noise_ratio=0.2):
array = np.copy(array)
for i in range(len(array)):
    for j in range(len(array[i])):
        if random.random() < noise_ratio:
            array[i][j] *= -1
    return array

def is_equal(true, preds):
eq = []
for i in range(len(preds)):
    eq.append(np.array_equal(preds[i], true[i]))
return eq

def evaluate_noise(noise_ratio, hebb_model, dataset):
eqs = []
for i in range(100000):
    noisy_inputs = add_noise(dataset, noise_ratio=noise_ratio)
    preds = [hebb_model.predict(i) for i in noisy_inputs]
    eqs.append(is_equal(dataset, preds))
draw_images(noisy_inputs, title="Noisy Inputs")
draw_images(preds, title="preds")
print(np.array(eqs).shape)
class_acc = np.sum(eqs, axis=0) / len(eqs)
print(f"Noise {noise_ratio * 100}% - Average Accuracy (over 100000 runs): {class_acc*100}%")

evaluate_noise(0.2, hebb, dataset)
evaluate_noise(0.8, hebb, dataset)
```



شکل 10 - نتیجه اضافه کردن نویز

همانطور که دیده می شود نویز 20 درصد قابل تحمل است و دقتهای حدود 90 درصد در تستهای مختلف توانستیم به دست آوریم. اما نویز 80 درصد مخصوصا به این شکل بایبولار، انگار که اکثر تصویر معکوس شود. به همین دلیل دقتها کاملا نزدیک 0 است و در مثال هم دیده می شود که به نظر اعداد حضور دارند ولی کاملا رنگشان برعکس شده است.

(ب)

همه اعداد به یک اندازه به نویز حساس نیستند. علت در تصاویر آمده است. دقت به تفکیک کلاس گزارش شده است و دیده می شود که به ترتیب برابر 90.1 و 95.8 و 89.6 است. به این ترتیب مقاومت ترین عدد 1 است. و همچنین 0 و 2 سخت تر هستند. حساس ترین عدد طبق دقتها، عدد 2 است. اگر دقت شود هم دیده می شود که بین این سه عدد، 0 و 2 خیلی شبیه هم هستند. مثلاً اگر همان خط وسط شکل 2 دچار نویز شود خیلی شبیه 0 می شود. (البته برعکسش سخت تر است) می توان یکی از دلایل حساس بودن 2 (و با اختلاف کمی بعد از آن 0) را همین دانست که با 0 خیلی مشابه هم هستند.

4) قسمت الف. گام قبل را برای حالتی که داده ها از بین رفته باشند (به جای مقادیر 1-9 صفر قرار گیرد) تکرار کنید.

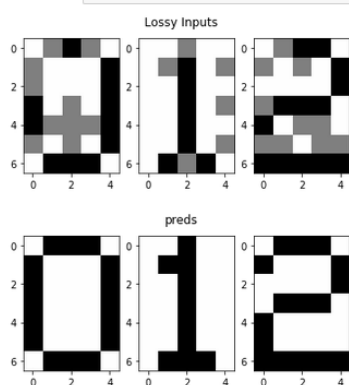
اضافه کردن از بین رفتن داده و نتایج آن در زیر آمده است.

4) Loss

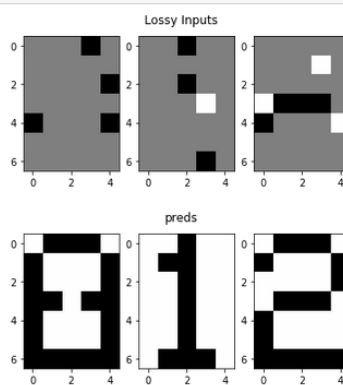
```
In [106]: def add_loss(array, loss_ratio=0.2):
    array = np.copy(array)
    for i in range(len(array)):
        for j in range(len(array[i])):
            if random.random() < loss_ratio:
                array[i][j] = 0
    return array

def evaluate_loss(loss_ratio, hebb_model, dataset):
    eqs = []
    for i in range(100000):
        lossy_inputs = add_loss(dataset, loss_ratio)
        preds = [hebb_model.predict(i) for i in lossy_inputs]
        eqs.append(is_equal(dataset, preds))
    draw_images(lossy_inputs, title="Lossy Inputs")
    draw_images(preds, title="preds")
    class_acc = np.sum(eqs, axis=0) / len(eqs)
    print(f"Loss {loss_ratio * 100}% - Average Accuracy (over 100000 runs): {class_acc*100}%")

evaluate_loss(0.2, hebb, dataset)
evaluate_loss(0.8, hebb, dataset)
```



Loss 20.0% - Average Accuracy (over 100000 runs): [99.995 100.



99.979]% Loss 80.0% - Average Accuracy (over 100000 runs): [74.391 84.354 69.019]%]

شکل 11 - نتیجه از بین رفتن داده

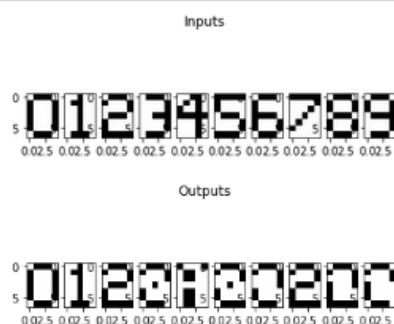
همانطور که دیده می شود، مدل نسبت به از بین رفتن داده مقاوم تر است و برای 20 درصد دقتهای حدود 99 و برای 80 درصد از دست رفتن داده دقت هایی بین 69 تا 84 درصد گرفته

است. همچنین می‌توان دید که در این بخش نیز، 0 و 2 بیشتر از 1 به مشکل خورده اند و مشخصا تشخیص 1 انقدر مجزا از بقیه بوده (احتمالا بخاطر خطی که در وسط به صورت عمودی دارد و بقیه ندارند.) که در 20 درصد از دست رفتن هیچگاه اشتباه نشده و 100 درصد درست تشخیص داده شده است.

5) (امتیازی) در این گام می‌خواهیم تعداد تصاویر را افزایش دهیم. بدین منظور به سراغ پوشه Extra بروید. اگر بررسی کنید مشاهده می‌فرمایید که الگوریتم‌های ذکر شده در گام قبل کارایی مطلوبی را از خود نشان نمی‌دهند. در این گام از شما می‌خواهیم روش شبه معکوس را پیاده کرده و قدرت به خاطر سپاری شبکه برای تصاویر موجود در پوشه Extra را با تکرار گام 3 مورد بررسی قرار دهید. روش را به مختصر شرح دهید. (توجه نمایید به توضیح الگوریتم بدون پیاده سازی یا پیاده سازی بدون توضیح مختصر الگوریتم نمره ای تعلق نمی‌گیرد.)

در روش Pseudo-inverse learning rule، مسئله را به شکل حل معادله چند مجهولی می‌بینیم و آن را به شکل ماتریسی حل می‌کنیم. یعنی می‌دانیم برای ورودی X باید خروجی هم پس از ضرب در ماتریس وزن‌ها W برابر X شود. $WX = X$ بنابراین کافی است این مسئله را حل کنیم که حل آن دقیقاً روش شبه معکوس است که ماتریسی را می‌دهد (W^+) که با ضرب در X ، جواب مسئله که باید همان X باشد را خروجی دهد. (البته روش اصلی لزوماً ورودی و خروجی یکسان نیستند و می‌توانند دو چیز متفاوت باشند ولی در این مسئله برای ما هر دو X هستند). نشان داده شده است که این روش گنجایش بیشتری نسبت به روش عادی هب دارد که در ادامه در نتایج هم می‌بینیم. بنابراین وزن‌ها را به شکل $W = X^+Y$ آموزش می‌دهیم. ابتدا نتیجه حالت هب قبلی را می‌بینیم.

```
In [121]: preds = [modified_hebb.predict(d) for d in dataset]
draw_images(dataset, title="Inputs")
draw_images(preds, title="Outputs")
```



```
In [89]: evaluate_noise(0.2, hebb, dataset)
evaluate_noise(0.8, hebb, dataset)
```



شکل 12 - نتیجه هب برای کلاسه‌های بیشتر

همانطور که دیده می‌شود این مدل به سختی عمل می‌کند و حتی بدون نویز هم نتوانسته بازسازی درستی داشته باشد. در نویز 20 درصد تنها 17.98 دقت داشته و در 80 درصد نویز هم هیچکدام را درست نگفته. (البته باید دقت داشت که نویز 80 درصد به معکوس تصویر نزدیک است و مدل نیز به معکوس آن همگرا می‌شود که کاملاً برعکس است)

اما در ادامه روش شبه معکوس را می‌بینیم.

```

In [117]: class PseudoinverseMabbTrainer:
def __init__(self, input_size: int, output_size: int):
    self.weights = np.zeros((input_size, output_size))

def train(self, inputs, outputs):
    self.weights = np.linalg.pinv(inputs) @ outputs

def predict(self, i):
    i = i.reshape(1, -1)
    pred = (i @ self.weights).reshape(-1)
    pred[pred > 0] = 1
    pred[pred < 0] = -1
    return pred

def draw_weights(self):
    mat = self.weights
    plt.imshow(mat, cmap=plt.cm.hot, figure=False)
    for (x, y), value in np.ndenumerate(mat.T):
        plt.text(x, y, f'{value:.0F}', va="center", ha="center", color="green")
    plt.title(f'Mabb Modified = {self.modified}')

In [118]: pinv = PseudoinverseMabbTrainer(dataset.shape[1], dataset.shape[1])
pinv.train(dataset, dataset)

In [119]: preds = [pinv.predict(d) for d in dataset]
draw_images(dataset, title="Inputs")
draw_images(preds, title="Outputs")

Inputs

0 0123456789
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025
Outputs

0 0123456789
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025

In [119]: evaluate_noise(0.1, pinv, dataset)
evaluate_noise(0.8, pinv, dataset)

Noisy Inputs

0 0023325333
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025
preds

0 0163169325
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025

(100000, 10)
Noise 20.0% - Average Accuracy (Over 100000 runs): [19.57 18.72 21.9 23.04 20.19 20.81 23.8 17.81 24.01 23.94]% -> 21.36
Noisy Inputs

0 0023333333
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025
preds

0 0163169325
5 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025 0.025

(100000, 10)
Noise 80.0% - Average Accuracy (Over 100000 runs): [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.00] -> 0.00

```

شکل 13- نتیجه روش شبه معکوس برای کلاسهای بیشتر

مهمترین خط در مدل، استفاده از `np.linalg.pinv` است که شبه معکوس را محاسبه می کند و با استفاده از آن در تست می توان در ورودی ضرب و خروجی مناسب را به دست آورد. همانطور که دیده می شود ابتدائاً تمام کلاس ها در حالت بدون نویز به درستی بازنمایی شدند. همچنین در نویز 20 درصد می بینیم که توانستیم دقت بهتری 21.36 نسبت به حالت قبل بگیریم. (همانطور

که قبلتر توضیح داده شد نویز 80 درصد به معکوس عکس همگرا می شود و در این حالت هم به همان علت دقت 0 می شود.) در مجموع دیدیم که این روش ظرفیت بیشتری نسبت به روش هب عادی دارد.

1) در مورد Discrete Hopfield Net مختصر توضیح دهید.

الگوریتم هاپفیلد همانطور که در صفحه 137 کتاب آمده آورده شده است.

Application Algorithm for the Discrete Hopfield Net

Step 0. Initialize weights to store patterns.

(Use Hebb rule.)

While activations of the net are not converged, do Steps 1–7.

Step 1. For each input vector x , do Steps 2–6.

Step 2. Set initial activations of net equal to the external input vector x :

$$y_i = x_i, (i = 1, \dots, n)$$

Step 3. Do Steps 4–6 for each unit Y_i .
(Units should be updated in random order.)

Step 4. Compute net input:

$$y_in_i = x_i + \sum_j y_j w_{ji}.$$

Step 5. Determine activation (output signal):

$$y_i = \begin{cases} 1 & \text{if } y_in_i > \theta_i \\ y_i & \text{if } y_in_i = \theta_i \\ 0 & \text{if } y_in_i < \theta_i. \end{cases}$$

Step 6. Broadcast the value of y_i to all other units.
(This updates the activation vector.)

Step 7. Test for convergence.

شکل 14 - الگوریتم هاپفیلد

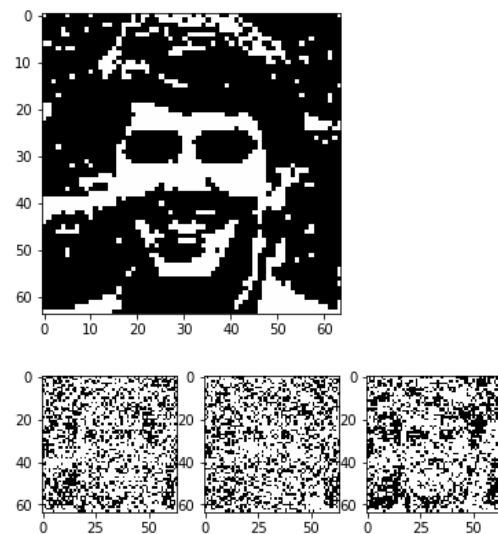
این روش جزء روش‌های iterative است به این معنا که شبکه چندین بار اجرا می‌شود تا همگرا شود. اما این روش خاص، تفاوت‌هایی دارد مانند اینکه هر یونیت به تنهایی آپدیت می‌شود و سپس سراغ یونیت بعدی می‌رویم (استپ 3). همچنین هر یونیت علاوه بر مقدار باقی یونیت‌ها، سیگنال خارجی را نیز می‌گیرد. ضمناً باید اشاره کرد که این روش با در نظر داشتن تابع انرژی طراحی شده و تکرار آن باعث همگرایی و کمینه شدن (لوکال) تابع انرژی می‌شود.

2) ابتدا سایز عکس ها را به 64×64 در بیاورید و سپس تصویر حاصل را به فرم Bipolar در بیاورید. (یعنی یک threshold بگذارید و پیکسل هایی با مقدار بیشتر از threshold مقدار 1 پیکسل هایی با مقدار کمتر از threshold مقدار -1 نسبت دهید). عکس حاصل را در گزارش قرار دهید. (نکته: مقادیر threshold مختلف تست کنید. این threshold مقدار عددی است بین 0 تا 255)

کد تغییر سایز و بایپولار کردن عکس و نتایج آن در زیر آمده است. برای ترشولد 130 انتخاب شد که کمی از 127 که وسط است بیشتر است و به نظر نتایج کمی بهتر داشت.

```
def load_image(img_path, threshold=130):
    img = Image.open(img_path)
    img.load()
    img = img.resize((64, 64))
    data = np.asarray(img, dtype="int32")
    data = np.mean(data, axis=-1) # RGB -> Grayscale
    data[data < threshold] = -1
    data[data >= threshold] = 1 # Bipolar
    data = data.flatten()
    return data

train, test = [], []
for i in range(1, 4, 1):
    test.append(load_image(f"Images_Q3/test{i}.png"))
train = load_image(f"Images_Q3/train.jpg").reshape(1, -1)
test = np.array(test)
draw_images(train)
draw_images(test)
```



شکل 15 - خواندن عکسها

3) ماتریس وزن ها را بر اساس تصویر قسمت قبلی بسازید.

کد پیاده سازی شده در زیر آمده است.

```
In [131]: class HopfieldTrainer:
def __init__(self, input_size: int, output_size: int, modified=False, threshold=0):
    self.weights = np.zeros((input_size, output_size))
    self.modified = modified
    self.threshold = threshold
    self.train_img = None

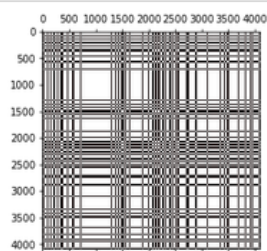
def train(self, inputs, outputs):
    self.train_img = outputs.reshape(1, -1)
    for i, o in zip(inputs, outputs):
        i = i.reshape(1, -1)
        o = o.reshape(1, -1)
        self.weights += i.T @ o
    # Modification
    if self.modified:
        np.fill_diagonal(self.weights, 0)

def predict(self, i, max_iter=2):
    i = i.reshape(1, -1) # (1, 4096)
    plt.figure(figsize=(2, 2))
    draw_images(i)
    y = np.copy(i) # Set initial activations of net equal to the external input vector x
    history = {"y": [], "hamming_distance": []}
    for iteration in tqdm(range(max_iter)):
        if len(history["hamming_distance"]) > 0 and history["hamming_distance"][-1] == 0:
            break
        print(f"### Iteration {iteration} ###")
        for enum, y_idx in enumerate(np.random.permutation(np.arange(i.shape[-1]))):
            y_in = i[0][y_idx] + np.dot(y[0], self.weights[:, y_idx])
            if y_in > self.threshold:
                y[0][y_idx] = 1
            elif y_in < self.threshold:
                y[0][y_idx] = -1
            if enum % 1000 == 0:
                print("Hamming Distance:", )
                history["y"].append(y)
                history["hamming_distance"].append(self.hamming_distance(y[0], self.train_img[0]))
    plt.figure(figsize=(16, 10))
    draw_images(history["y"], subtitles=np.round(history["hamming_distance"], 2))
    return y

def hamming_distance(self, a1, a2):
    return np.sum(a1 != a2) / len(a1)

def draw_weights(self):
    mat = self.weights
    plt.matshow(mat, cmap=plt.cm.hot, fignum=False)
    for (x, y), value in np.ndenumerate(mat.T):
        # plt.text(x, y, f"{value:.0f}", va="center", ha="center", color="green")

In [132]: hofield = HopfieldTrainer(train.shape[1], train.shape[1], modified=False)
hofield.train(train, train)
hofield.draw_weights()
```



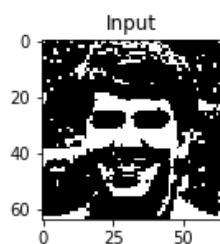
شکل 16 - پیاده سازی هاپفیلد

همانطور که در الگوریتم هم آمده است مرحله مشخص شدن وزن‌ها با همان روش هب انجام می‌شود. وزن‌های به دست آمده در شکل نمایش داده شده است.

4) با کمک ماتریس وزن های قسمت 3 سعی کنید تصویر اصلی را بازیابید. در هر 50 iteration عکس حاصل را چاپ کنید.

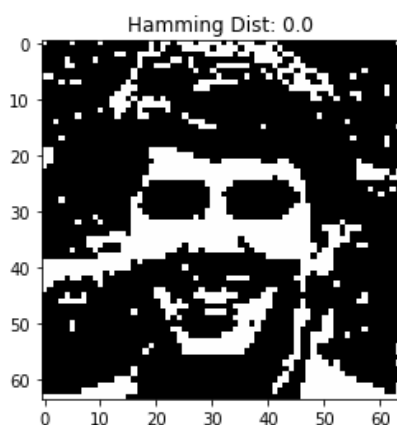
نتیجه اجرای الگوریتم روی ورودی اصلی به شکل زیر است.

```
: preds = [hofield.predict(d, log_steps=1) for d in train]
# draw_images(train, title="Inputs")
# draw_images(preds, title="Outputs")
```



0% 0/2 [00:00<?, ?it/s]

Iteration 0



شکل 17 - نتیجه اجرا روی ورودی اصلی

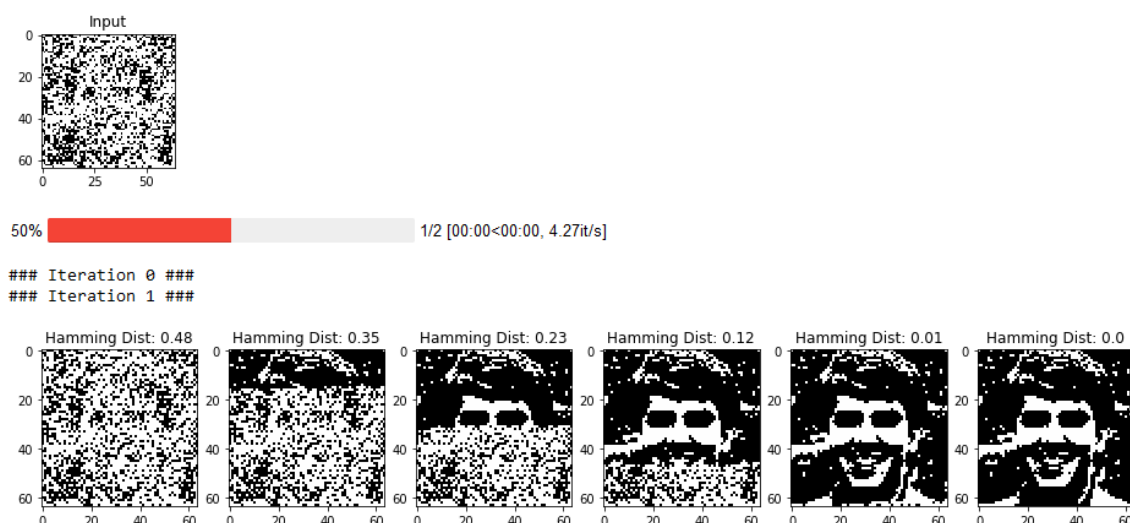
همانطور که دیده می شود با آپدیت اولین یونیت همگرا می شویم و hamming distance برابر 0 داریم. در اصل همان مقدار دهی اولیه وزن ها با استفاده از هب توانسته ورودی را به خروجی برساند و نیازی به تکرار نبوده است.



شکل 18 - نتیجه اجرا روی ورودی نویزدار

در این مورد هم می بینیم که سرعت همگرایی بالا است و اصلا به یک iteration کامل نمیرسیم و hamming distance برابر 0 می شود. همچنین تصاویر مراحل این اتفاق نمایش داده شده است که دیده می شود چگونه تصویر اصلاح می شود.

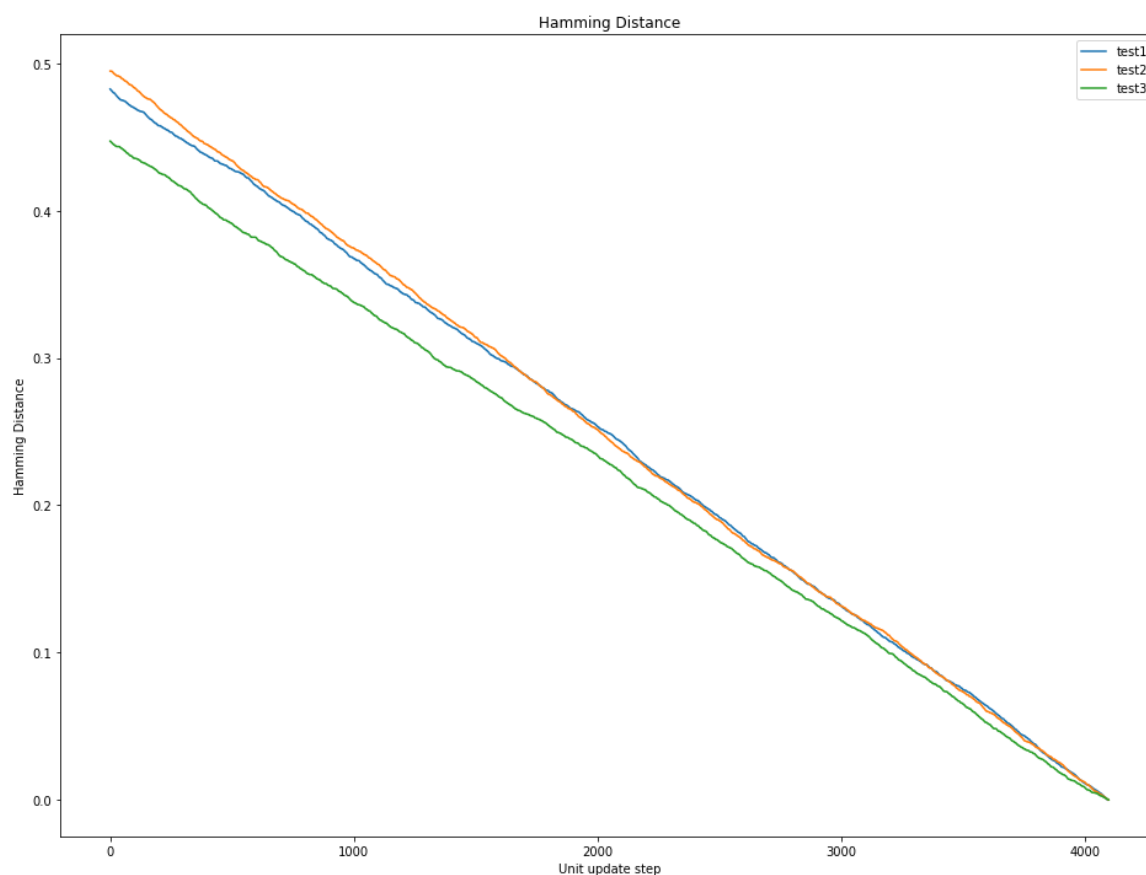
برای بهتر مشخص شدن موضوع یک بار هم بدون اینکه طبق الگوریتم به صورت تصادفی نوروں ها را آپدیت کنیم به ترتیب این کار را می کنیم.



شکل 19 - نتیجه اجرا روی ورودی نویزدار بدون ترتیب تصادفی

و دیده می شود که نوروںها به ترتیب تاثیر داده می شوند و تصویر ساخته می شود و نیازی به iteration دوم هم نیست.

5) نمودار Hamming Distance per iteration هر تصویر تست را رسم کنید.



شکل 20 - نتیجه hamming distance در طول اجرای الگوریتم

در این نمودار همانطور که دیده می شود فاصله در حال کمتر شدن است و در انتها 0 می شود. همچنین می توان دید که طبق تصویرهایی که داده شده بود شماره 3 کمترین نویز (و در نمودار هم به طور نسبی کمترین فاصله همینگ را دارد) و شماره 2 بیشترین نویز را داشته و بیشترین فاصله در نمودار دارد.

سوال 4 – Bidirectional Associative Memory

1) ماتریس وزن را بدست آورید و در گزارش مکتوب کنید

در ابتدا توابعی برای تبدیل استرینگ به باینری و باینری به استرینگ نوشتیم که در زیر آمده است.

```
def to_binary(s: str):
    b = bin(int.from_bytes(s.encode(), 'big'))
    b_list = [int(bi) for bi in b[2:]]
    return np.array(b_list)

def to_str(b_array):
    b = "".join([str(n) for n in b_array])
    b = "0b" + b
    n = int(b, 2)
    return n.to_bytes((n.bit_length() + 7) // 8, 'big').decode()

b = to_binary("Clinton")
s = to_str(b)
print(b)
print(s)
```

[1 0 0 0 0 1 1 0 1 1 0 1 1 0 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 1
 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 0]
 Clinton

شکل 21 - توابع تبدیل

این مدل به شکل زیر پیاده سازی شد:

```
class BamTrainer:
    def __init__(self, input_size: int, output_size: int, modified=False, threshold=0):
        self.weights = np.zeros((input_size, output_size))
        self.modified = modified
        self.threshold = threshold
        self.input_size = input_size
        self.output_size = output_size

    def train(self, inputs, outputs):
        for i, o in zip(inputs, outputs):
            i = i.reshape(1, -1)
            o = o.reshape(1, -1)
            self.weights += i.T @ o
        # Modification
        if self.modified:
            np.fill_diagonal(self.weights, 0)

    def predict(self, i, pred_y=True, max_iter=2, log_steps=1000, save_images=True):
        def update_x(x, y):
            for y_idx in np.arange(y.shape[-1]):
                y_in = np.dot(x[0], self.weights[:, y_idx])
                if y_in > self.threshold:
                    y[0][y_idx] = 1
                elif y_in < self.threshold:
                    y[0][y_idx] = -1

        def update_y(x, y):
            for x_idx in np.arange(x.shape[-1]):
                x_in = np.dot(y[0], self.weights[x_idx, :])
                if x_in > self.threshold:
                    x[0][x_idx] = 1
                elif x_in < self.threshold:
                    x[0][x_idx] = -1

        visited_patterns = []
        i = i.reshape(1, -1)
        if pred_y:
            x = np.copy(i) # Present input pattern x to the X-layer
            y = np.zeros((1, self.output_size))
        else:
            y = np.copy(i) # Present input pattern x to the X-layer
            x = np.zeros((1, self.input_size))
        for iteration in tqdm(range(max_iter)):
            print(f"### Iteration {iteration} ###")
            if pred_y:
                # Update activations of units in Y-layer
                update_y(x, y)
                # Update activations of units in X-layer
                update_x(x, y)
            output = y
```

```

else:
    update_x(x, y)
    update_y(x, y)
    output = x
# Test for convergence:
if str(output.flatten()) in visited_patterns:
    break
else:
    visited_patterns.append(str(output.flatten()))
return output.flatten()

def draw_weights(self):
    mat = self.weights
    plt.figure(figsize=(10, 7))
    plt.matshow(mat, cmap=plt.cm.hot, figure=False)
    for (x, y), value in np.ndenumerate(mat.T):
        plt.text(x, y, f"{{value:.0f}}", va="center", ha="center", color="gray")

```

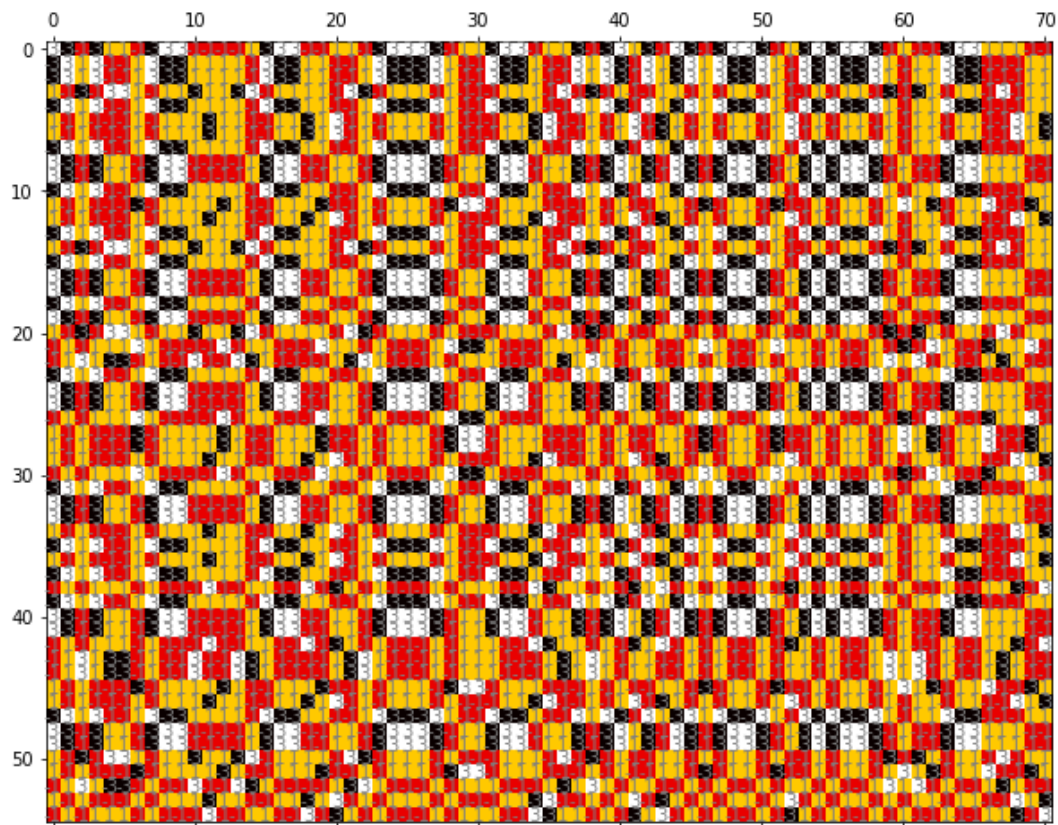
کد 1 - پیاده سازی هم

و نتیجه وزن ها در زیر آمده است.

```

bam = BamTrainer(X_b.shape[1], Y_b.shape[1], modified=False)
bam.train(X_b, Y_b)
bam.draw_weights()

```





شکل 22 - ماتریس وزن


2) کارایی شبکه در بازیابی اطلاعات از هر دو جهت را بررسی کنید و نتایج را به طور کامل گزارش نمایید.

در زیر نتیجه تست هر دو طرف آمده است.


```
def accuracy(a1, a2):  
    return np.sum(a1 == a2) / len(a1) * 100  
  
preds = np.array([bam.predict(x, pred_y=True) for x in X_b])  
for i in range(3):  
    print("Input:", to_str(X_b[i]), "| Pred:", to_str(preds[i]), "| Accuracy:", accuracy(Y_b[i], preds[i]))  
  
preds = np.array([bam.predict(y, pred_y=False) for y in Y_b])  
for i in range(3):  
    print("Input:", to_str(Y_b[i]), "| Pred:", to_str(preds[i]), "| Accuracy:", accuracy(X_b[i], preds[i]))
```


50%  1/2 [00:00<00:00, 43.48it/s]


50%  1/2 [00:00<00:00, 50.01it/s]

50%  1/2 [00:00<00:00, 47.57it/s]

Input: Clinton | Pred: President | Accuracy: 100.0
Input: Hillary | Pred: FirstLady | Accuracy: 100.0
Input: Kenstar | Pred: Gentleman | Accuracy: 100.0

50%  1/2 [00:00<00:00, 43.48it/s]

50%  1/2 [00:00<00:00, 43.47it/s]

50%  1/2 [00:00<00:00, 43.48it/s]

Input: President | Pred: Clinton | Accuracy: 100.0
Input: FirstLady | Pred: Hillary | Accuracy: 100.0
Input: Gentleman | Pred: Kenstar | Accuracy: 100.0

شکل 23 - نتیجه اجرای بام

همانطور که دیده می شود دقت هر دو طرف به راحتی 100 درصد شده است و همچنین باید اشاره کرد که در همان اولین بار اجرای الگوریتم این اتفاق افتاده و نیازی به تکرار بیشتر نبوده است.

3) در این گام به صورت تصادفی ابتدا 10% بیتها و سپس 20% بیتها برای هر یک از ورودیها در هر دو جهت را نویزی کرده و درصد خروجی درست شبکه را گزارش کنید. (دقت کنید که در این گام میبایست تشابه را بر اساس تعداد بیتهای برابر خروجی شبکه و خروجی مطلوب بر حسب درصد گزارش کنید. برای آنکه نتایج شما قابل تعمیم باشد، می بایست در یک حلقه ی صدتایی این میزان را بررسی کنید و نهایتاً 6 عدد را به ازای 10% نویز تصادفی برای رفت و برگشت و نیز 6 عدد به ازای 20% نویز تصادفی برای رفت و برگشت بر حسب درصد در یک جدول ارائه کنید. همچنین برای اعمال نویز کافی است به جای 1، -1 و به جای -1، 1 قرار دهید.)

کد و نتیجه اجرا در زیر آمده است.

```
def add_noise(array, noise_ratio=0.2):
    array = np.copy(array)
    for i in range(len(array)):
        for j in range(len(array[i])):
            if random.random() < noise_ratio:
                array[i][j] *= -1
    return array

def is_equal(true, preds):
    eq = []
    for i in range(len(preds)):
        eq.append(np.array_equal(preds[i], true[i]))
    return eq

def evaluate_noise_x_to_y(noise_ratio, bam):
    class_acc = [[], [], []]
    for i in range(100):
        noisy_inputs = add_noise(X_b, noise_ratio=noise_ratio)
        preds = [bam.predict(x, pred_y=True) for x in noisy_inputs]
        for i in range(len(preds)):
            class_acc[i].append(accuracy(Y_b[i], preds[i]))
    print(f"Noise {noise_ratio * 100}% - Average Accuracy (over 100 runs): {np.array(class_acc).mean(axis=-1)}% -> {np.mean(class_acc)}")

def evaluate_noise_y_to_x(noise_ratio, bam):
    class_acc = [[], [], []]
    for i in range(100):
        noisy_inputs = add_noise(Y_b, noise_ratio=noise_ratio)
        preds = [bam.predict(y, pred_y=False) for y in noisy_inputs]
        for i in range(len(preds)):
            class_acc[i].append(accuracy(X_b[i], preds[i]))
    print(f"Noise {noise_ratio * 100}% - Average Accuracy (over 100 runs): {np.array(class_acc).mean(axis=-1)}% -> {np.mean(class_acc)}")

evaluate_noise_x_to_y(0.1, bam)
evaluate_noise_x_to_y(0.2, bam)
evaluate_noise_y_to_x(0.1, bam)
evaluate_noise_y_to_x(0.2, bam)
```

```
Noise 10.0% - Average Accuracy (over 100 runs): [98.90140845 99.85915493 99.15492958]% -> 99.31%
Noise 20.0% - Average Accuracy (over 100 runs): [96.33802817 99.15492958 96.78873239]% -> 97.43%
Noise 10.0% - Average Accuracy (over 100 runs): [ 99.70909091 100.          99.70909091]% -> 99.81%
Noise 20.0% - Average Accuracy (over 100 runs): [98.83636364 97.8          97.81818182]% -> 98.15%
```

شکل 24 - نتیجه اجرای بام با نویز

همانطور که دیده می شود به ازای هر کلاس و در هر دو جهت تغییری که خواسته شده داده شده و نتیجه 100 بار اجرا میانگین گرفته شده است. این نتایج در جدول زیر نیز آمده اند.

```
import pandas as pd
pd.DataFrame([[98.90140845, 99.85915493, 99.15492958],
              [96.33802817, 99.15492958, 96.78873239],
              [ 99.70909091, 100., 99.70909091],
              [98.83636364, 97.8, 97.81818182]
              ],
              columns=["Clinton/President", "Hillary/FirstLady", "Kenstar/Gentleman"],
              index=["x_to_y 10% Noise", "x_to_y 20% Noise", "y_to_x 10% Noise", "y_to_x 20% Noise"]
              )
```

	Clinton/President	Hillary/FirstLady	Kenstar/Gentleman
x_to_y 10% Noise	98.901408	99.859155	99.154930
x_to_y 20% Noise	96.338028	99.154930	96.788732
y_to_x 10% Noise	99.709091	100.000000	99.709091
y_to_x 20% Noise	98.836364	97.800000	97.818182

شکل 25 - نتیجه اجرای هم با نویز به شکل جدول

همانطور که دیده می‌شود، مقاومت مدل نسبت به نویز 10 و 20 درصد در مجموع مناسب و بالا است. اما طبیعتاً در نویزهای 20 درصد دقتها کمتر از نویز 10 درصد است. ضمناً به نظر می‌رسد که از y به x دقتهای بهتری داریم که البته چون طول y نسبت به x بلندتر است طبیعتاً گنجایش بیشتری دارد و بعد از نویز احتمالاً می‌تواند باز اطلاعات بیشتری در خود نگه دارد و از سمت بزرگ به کوچک برای مدل راحت تر است.

4) حال یک شخصیت دیگر را در کنار سه شخصیت قبلی در آموزش شرکت دهید و بررسی کنید چه تعداد از خروجیها توسط ورودیها و چه تعداد از ورودیها توسط خروجیها قابل بازیابی است؟ آیا کارایی شبکه کاهش یافته است یا خیر؟ دلیل خود را شرح دهید.

نتیجه اضافه کردن شخصیت جدید در زیر آمده است.

```
X = ["Clinton", "Hillary", "Kenstar", "Lewisky"]
Y = ["President", "FirstLady", "Gentleman", "SweetGirl"]
X_b = np.array([to_binary(s) for s in X]) * 2 - 1
Y_b = np.array([to_binary(s) for s in Y]) * 2 - 1
# X_b
```

```
def accuracy(a1, a2):
    return np.round(np.sum(a1 == a2) / len(a1) * 100, 2)

bam = BamTrainer(X_b.shape[1], Y_b.shape[1], modified=False)
bam.train(X_b, Y_b)

preds = np.array([bam.predict(x, pred_y=True) for x in X_b])
for i in range(4):
    print("Input:", to_str(X_b[i]), "| Pred:", to_str(preds[i]), "| Accuracy:", accuracy(Y_b[i], preds[i]))

preds = np.array([bam.predict(y, pred_y=False) for y in Y_b])
for i in range(4):
    print("Input:", to_str(Y_b[i]), "| Pred:", to_str(preds[i]), "| Accuracy:", accuracy(X_b[i], preds[i]))
```

Input: Clinton	Pred: Rsesldefl	Accuracy: 91.55
Input: Hillary	Pred: FafstDadl	Accuracy: 91.55
Input: Kenstar	Pred: Gefulem`l	Accuracy: 94.37
Input: Lewisky	Pred: SweutEibl	Accuracy: 95.77
Input: President	Pred: Kmmntkz	Accuracy: 89.09
Input: FirstLady	Pred: Hmllqcy	Accuracy: 92.73
Input: Gentleman	Pred: Kenktcz	Accuracy: 92.73
Input: SweetGirl	Pred: Heoiqky	Accuracy: 92.73

شکل 26 - نتیجه اجرای بام با یک ورودی جدید

همانطور که دیده می‌شود دقت تمام موارد و حالتها از 100 به حدود 90 رسیده است. بنابراین هیچ کدام به صورت کامل قابل بازیابی نیستند و صرفاً حدود 90 درصد آنها بازیابی شده است. از نظر اندازه ذخیره سازی یا Storage Capacity همانطور که در صفحه 149 کتاب آمده است، نباید تعداد association هایی که ذخیره می‌کنیم بیشتر از تعداد نورون‌های لایه کوچک‌تر شود. همچنین باید طبق هب توجه داشت که اگر ورودی‌ها متعامد باشند مدل می‌تواند قطعا بازیابی کند. بنابراین ترکیب این دلایل و ظرفیت ذخیره‌سازی محدود شبکه باعث می‌شود با اضافه کردن ورودی جدید، دیگر مدل نتواند به خوبی بازیابی را انجام دهد. بنابراین همانطور که دیده می‌شود مدل در بازیابی دچار اختلال می‌شود و توانایی بازشناسی درست و کامل را از دست می‌دهد.