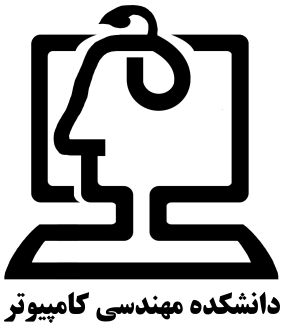
به نام خدا



دانشگاه صنعتی امیرکبیر
دانشکده مهندسی کامپیوتر

استاد درس: دکتر صفابخش

بهار ۱۴۰۰

درس شبکه های عصبی مصنوعی

سروش مهدی
شماره دانشجویی: ۹۹۱۳۱۰۵۰

گزارش تمرین سوم

فهرست مطالب

۱ ساختار شبکه خودسازمانده کوهونن

این شبکه یک نوع شبکه عصبی بدون ناظر هست که از رقابت بین نورون ها در مغز الهام گرفته شده است. در این شبکه نورون برنده و همسایه هایش به طوری وزن های خود را تغییر میدهند که به نوع خاصی از داده ها حساس تر باشند.
این همسایگی که به صورت های مختلف تعریف میشود باعث میشود که خواص توپولوزی داده ها نیز در نظر گرفته و حفظ شود. همچنین نورون هایی که نزدیک تر باشند به نورون برنده وزنشان بیشتر تغییر میکند

این شبکه دو لایه دارد که لایه اول لایه ورودی میباشد و لایه دوم نورون هایی هستند که میخواهیم داده ها را روی ان ها تصویر کنیم و بازنمایی داده ها را در ابعادی احتمالا کوچک تر روی ان ها نمایش دهیم. لایه دوم را معمولا در دو بعد نمایش میدهند یعنی نورون ها به صورت دو بعدی کنار هم چیده شده اند.
برای بروز رسانی وزن ها ابتدا باید در هر مرحله یادگیری روی یک داده نورون برنده مشخص شود. این کار میتواند با در نظر گرفتن فاصله اقلیدسی وزن های هر نورون و هر ورودی انجام شود به این صورت که برای هر ورودی نورونی را که وزن هایش با ان ورودی کمترین فاصله را دارند به عنوان نورون برنده انتخاب میکنیم.
سپس همسایگی این نورون را محاسبه میکنیم و بر اساس نوع همسایگی که در نظر میگیریم نورون های دیگر هر کدام از این نورون برنده فاصله ای دارند که متناسب با عکس فاصله این نورون ها به علاوه نورون برنده وزنشان اپدیت شده به طوریکه به ورودی مورد نظر حساس تر شوند و به بیان دیگر فاصله ان ها از ورودی مورد نظر کمتر شود. فرمول این به روز رسانی به صورت زیر هست

$$w_i = w_i + \beta * NS * (x - w_i) \tag{1}$$

در این معادله بتا ضریب یادیگری و x بردار داده ورودی و w بردار وزن نورون مورد نظر و NS ضریب همسایگی نورون با نورون برنده میباشد.
برای کاهش حجم داده ها میتوان داده ها را به وسیله این شبکه از یک فضای پیوسته به یک فضای گسسته برد. در واقع میتوانیم عمل vector-quantization را با ین شبکه انجام داد.

۲ تفاوت خوشه بندی و کاهش ابعاد

برای خوشه بندی ابتدا U-matrix شبکه را رسم میکنیم که در ان میانگین فاصله هر نورون از همسایه هایش مشخص میشود سپس ناحیه هایی را که در این نقشه نورون هایش نزدیک به هستند میتوانیم به عنوان خوشه ها در نظر بگیریم و ببینیم هر داده به کدام خوشه مپ میشود. و یا میتوان هر نورون را به عنوان یک خوشه در نظر گرفت.
در مورد کاهش ابعاد داده نیز میتوانیم برای هر داده مختصات نورونی که به ان مپ میشود یا وزن ها ان نورون را به عنوان ویژگی ها بگیریم روش دیگر این هست که برای هر داده فاصله اش از هر نورون را به عنوان ویژگی های جدید در نظر بگیریم
شباهت این دو روش در این هست که در هر دو روش نحوه اموزش شبکه یکسان هست اما برای خوشه بندی و کاهش ابعاد از نقشه های مختلفی استفاده میکنیم. در واقع در خوشه بندی فاصله نورون ها از یکدیگر برای ما مهم هست اما در کاهش ابعاد نسبت هر داده به نورون ها برای ما مهم هست.

۳ پیاده سازی شبکه کوهونن

ابتدا یک تابع برای اماده کردن مجموعه های اموزش و تست و اعتبارسنجی مینویسیم که در زیر امده است.

```
def create_data_set():
    """ Load the Yale Faces data set and generate labels for each image.
        Returns: Train, test, validation samples with their labels and classes names The training samples are flat
        of size  (243*320) , the labels are one-hot-encoded values for each category
    """
    images_path = [ os.path.join("yalefaces", item)  for item in  os.listdir("yalefaces") ]
    image_data = []
    image_labels = []

    for im_path in images_path:
        #reading each image
        im = io.imread(im_path , as_gray = True)
        image_data.append(np.array(im, dtype='uint8'))
        #extracting labels
        label = os.path.split(im_path)[1].split(".")[1]

        image_labels.append(label)

    X_ = np.array(image_data).astype('float32')
    enc = LabelBinarizer()
    y_ = enc.fit_transform(image_labels)
    #splitting data
    X_train, X_test, y_train, y_test = train_test_split(X_, y_, train_size=0.7, random_state = 41)
    X_val , X_test , y_val , y_test = train_test_split(X_test, y_test, train_size=2/3, random_state = 41)
    #normalizing data
    X_test = X_test/255.0
    X_train = X_train/255.0
    X_val = X_val/255.0
    return (X_train).reshape((X_train.shape[0],X_train.shape[1]*X_train.shape[2])),\
            (X_test).reshape((X_test.shape[0],X_test.shape[1]*X_test.shape[2]))\
            ,X_val.reshape((X_val.shape[0],X_val.shape[1]*X_val.shape[2])), y_train, y_test , y_val , enc.classes

X_train , X_test, X_val, y_train , y_test , y_val , labels_names = create_data_set()
```

Listing : ۱

این تابع تمامی تصاویر را میخواند و برچسب ها را از روی اسم ان ها میسازد با فرمت one-hot ذخیره میکند و سپس داده ها را با نسبت خواسته شده تقسیم میکند . بعد از ان داده ها را در بازه صفر و یک نرمال میکند و در نهایت داده ها را که به صورت ماتریس هستند به بردار تغییر میدهد.
حال به بررسی کلاس SOM پیاده شده میپردازیم.
این کلاس توابع مختلفی دارد که به توضیح ان ها میپردازیم در صفحه بعد کد تابع init امده است . همینطور که مشاهده میشود این تابع به عنوان ورودی معماری شبکه و نرخ یادگیری و پارامتر سیگما و نرخ کاهش این پارامتر ها را میگیرد. سپس وزن های شبکه به صورت رندوم مقدار دهی میشوند همچنین دو ماتریس داریم که در یکی برای هر خانه شماره سطر و ستون را نگه میداریم و دیگری یک mask هست که به ازای هر خانه و با توجه به پارامتر سیگما اندازه همسایگی های ان خانه تا سایر خانه ها را دارد
در ادامه دو تابع امده اند که یکی نورن برنده را به ازای یک ورودی خاص مشخص میکند و دیگری برای

```python
class SOM:
    def __init__(self ,map_size , lr , sigma , decay):
        #map size = (w,h,f)
        self.map = np.random.random(map_size)
        #ind matrix is a matrix that holds index of each cell
        self.ind_matrix = np.zeros((self.map.shape[0] , self.map.shape[1] , 2))
        #mask matrix is a matrix that for each neuron holds a
        #mask that is corrosponding to gaussian neighborhood for given sigma
        self.mask = np.zeros((map_size[0] ,map_size[1] ,map_size[0] ,map_size[1]))
        self.lr = lr
        self.sigma = sigma
        self.decay = decay
        #initilizing ind matrix
        for i in range(map_size[0]):
            for j in range(map_size[1]):
                self.ind_matrix[i,j] = [i,j]
        #initializing mask
        for i in range(map_size[0]):
            for j in range(map_size[1]):
                self.mask[i][j] = self.get_N_mask(np.array([i,j]) , sigma)
```

Listing :۲

یک مختصات ورودی یک ماسک بر اساس همسایگی گاوسی میسازد. در این کلاس ما ابتدا این ماسک را برای همه خانه ها با توجه به سیگمای اولیه بدست می اوریم و به جای اینکه در هر گام دوباره این ماسک را محاسبه کنیم فقط ماسک قبلی را به توان معکوس توان دو مقداری که قرار است سیگما کاهش یابد ضرب میکنیم این کار برای بهتر شدن سرعت انجام شده و معادل کاهش سیگما به صورت پله ای هست.

```python
    def get_winner(self,x):
        #find winner for x vector
        dists = np.linalg.norm(self.map - x , axis = 2)
        winner = np.unravel_index(np.argmin(dists), dists.shape)
        return np.array(winner)

    def get_N_mask(self ,winner , sigma):
        #find gaussian mask for winner with given sigma
        mask = np.linalg.norm(self.ind_matrix - winner , axis = 2)
        mask = mask**2
        mask = -(mask/(2*sigma**2))
        mask = np.exp(mask)
        return mask
```

Listing :۳

در ادامه تابع اموزش امده است در این تابع مقادیر سیگما و نرخ یادگیری به صورت پله ای و هر ده تکرار یکبار کاهش میابند. برای کاهش این پارامتر ها روش های متخلفی امتحان کردم و این روش عملکرد بهتری داشت. همچنین برای محاسبات سعی شده کمتر بار محاسباتی را با استفاده از عملیات های ماتریسی داشته باشیم. در این تابع به تعداد هر بار روی داده های تصادفی فرایند اموزش را انجام میدهیم.

در ادامه دیگر توابع این کلاس امده است. در تابع u-matrix مقدار میانگین فاصله هر نورون از همسای

```
    def train(self,X_train  , y, itr_num , error_t = 10**-10):
        Js = []
        #k is decay power
        k = -1
        sigma = self.sigma
        beta = self.lr

        for i in range(itr_num):
            if i%10==0:
                k+=1
            prev_map = np.copy(self.map)
            shuffle_ind = np.random.randint(low = 0 , high = len(X_train) , size =len(X_train))
            sigma_change = 1/(self.decay**(2*k))
            beta = self.lr * self.decay**k

            for j in range(len(X_train)):
                x = X_train[shuffle_ind[j]]
                winner = self.get_winner(x)
                mask = np.power(self.mask[winner[0] , winner[1]] , sigma_change)
                self.map = self.map + beta*( mask[:,:,None] * (x - self.map))

            Js.append(np.linalg.norm(prev_map - self.map))

            if Js[-1]<error_t:
                return Js
        return Js
```

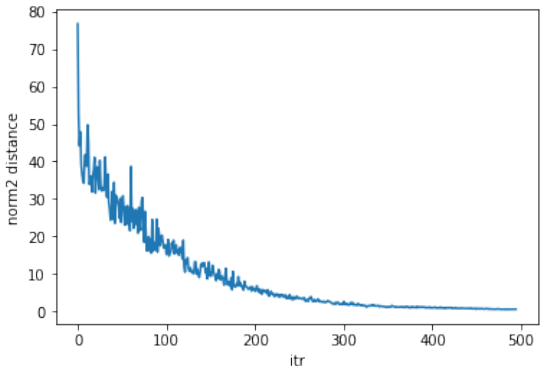
Listing :۴

های مربعی اش به فاصله یک محاسبه میشود. در تابع purity در هر خوشه کلاسی که بیشترین تعداد را دارد مشخص میشود و تعداد نمونه های ان کلاس در ان خوشه مشخص شده و این مقادیر جمع میشوند و حاصل بر تعداد کل داده تقسیم میشود. تابع اخر نیز برای بصری سازی توزیع داده ها میباشد.
در تمامی بخش های این تمرین مجموعه داده ها را طبق نسبت های گفته شده تقسیم کردم و شبکه SOM را فقط روی داده های اموزش اموزش دادم که شامل ۷۰ درصد داده ها بود.

```python
    def u_matrix(self):
        #calculating u matrix for square neighborhood
        u_matrix = np.zeros((self.map.shape[0],self.map.shape[1]))
        for i in range(u_matrix.shape[0]):
            for j in range(u_matrix.shape[1]):
                t = 0
                n = 0
                for k in [1,0,-1]:
                    for l in [1,0,-1]:
                        if (i+k) in range(u_matrix.shape[0]) and (j+l) in range(u_matrix.shape[1]):
                            t+= np.linalg.norm(self.map[i+k][j+l]-self.map[i][j])
                            n+=1
                u_matrix[i][j] = t/(n-1)
        return u_matrix

    def purity(self , X , y):
        #y is one hot
        map_size = self.map.shape
        count = np.zeros((map_size[0] , map_size[1] , y.shape[1]))
        for i in range(len(X)):
            x = X[i]
            winner = self.get_winner(x)
            count[winner[0] , winner[1]] += y[i]

        t = np.max(count , axis = 2)
        purity = np.sum(t) / len(X)
        return purity

    def visualize(self,X,y,  colors , label_names):

        w_x , w_y = zip(*[self.get_winner(d) for d in X])
        w_x = np.array(w_x)
        w_y = np.array(w_y)
        target = np.argmax(y , axis = 1)
        ax = plt.figure(figsize=(8, 8))
        plt.pcolor(self.u_matrix(), cmap='bone_r', alpha=.5)
        plt.colorbar()

        for c in np.unique(target):
            idx_target = target==c
            plt.scatter(w_x[idx_target]+.5+(np.random.rand(np.sum(idx_target))-.5)*.8,
                        w_y[idx_target]+.5+(np.random.rand(np.sum(idx_target))-.5)*.8,
                        s=30, color=colors[c], label=label_names[c] )

        plt.legend(loc='upper left',bbox_to_anchor=(1.2, 1.0))
        plt.grid()
        plt.show()
```

Listing :۵

در این بخش مقادیر مختلفی بین ۵ تا ۸ را برای طول و عرض شبکه امتحان کردم که در هر مورد بهترین مقدار purity با توجه به مقادیر مختلف شعاع همسایگی و نرخ یادگیری بین ۴۰ تا ۵۰ درصد بود. بهترین معماری معماری ۸ در ۸ بود و مقادیر شعاع همسایگی و نرخ یادگیری برای این معماری در بهترین حالت به ترتیب برابر ۳ و ۰/۰۲ بودند و همچنین پارامتر decay نیز ۰/۹۵ در نظر گرفته شد. در این حالت مقدار purity برابر ۵۰ درصد میشود. همچنین اندازه تغییرات وزن ها در تکرار های مختلف در زیر نشان داده شده که برای خوانایی بهتر از گام پنجم به بعد پلات شده است در این حالت purity مجموعه تست ۹۰ درصد میباشد که مقدار بالای این معیار به دلیل تعداد کم داده های تست میباشد. همچنین مقدار purity به ازای برخی مقادیر



شکل ۱: اندازه تغییرات وزن ها در هر گام

مختلف پارامتر ها به صورت زیر هست

SOM((8,8,77760) , lr = .08 ,sigma = 4. , decay = 0.9): 47 per cent purity
SOM((5,5,77760) , lr = .04 ,sigma = 2. , decay = 0.95): 43 per cent purity
SOM((6,6,77760) , lr = .08 ,sigma = 3. , decay = 0.9): 43 per cent purity
SOM((7,7,77760) , lr = .08 ,sigma = 4. , decay = 0.9): 45 per cent purity

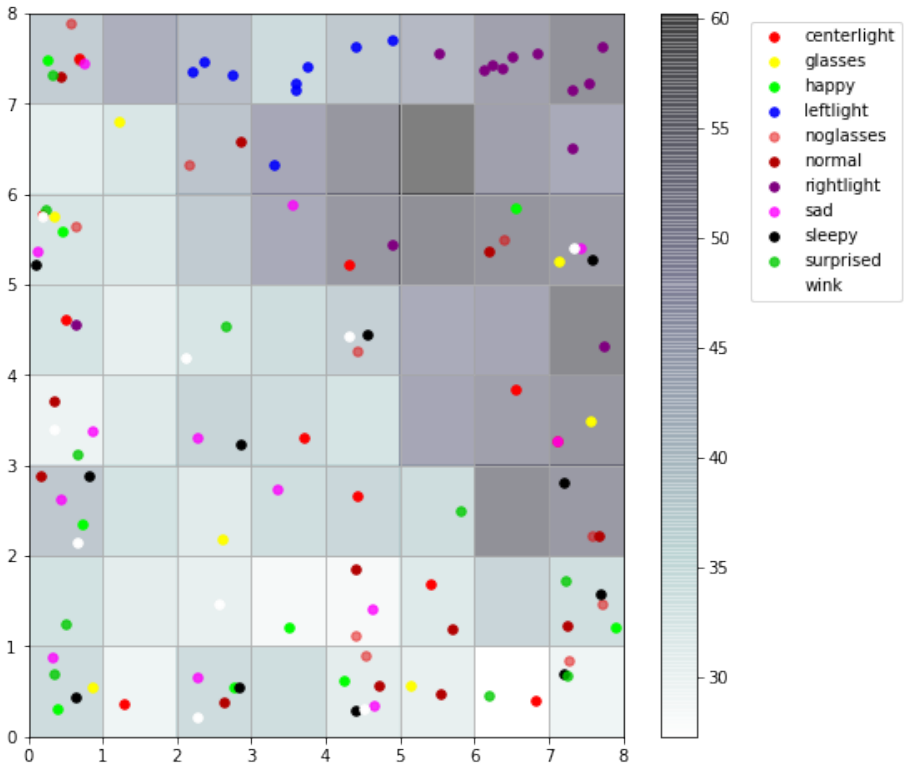
۴ کاهش ابعاد

برای کاهش ابعاد یک تابع به کلاس پیاده شده اضافه میکنیم که برای هر نمونه داده فاصله تا هر نورن را حساب میکند و این مقادیر را به عنوان ویژگی های جدید برمیگرداند. این تابع به صورت زیر میباشد

```
def extract_feature(self , X):
    features = np.zeros((len(X) , self.map.shape[0],self.map.shape[1]))
    for i in range(len(X)):
        x = X[i]
        features[i] = 1 / (1 + np.linalg.norm(self.map - x , axis = 2))
    return features
```

Listing :۶

توزیع بصری داده ها بعد از ۵۰۰ تکرار به صورت زیر میباشد در این شکل رنگ پس زمینه سلول ها نیز



شکل ۲: توزیع داده ها

u-matrix را مشخص میکند.

مشاهده میشود که شبکه دو کلاس leftlight و rightlight را میتواند به خوبی جدا کند با بررسی داده ها متوجه میشویم نوع نورپردازی در این دو کلاس بازه زیادی از داده ها را که در تمامی کلاس های دیگر سفید هستند تیره کرده. در واقع قسمت بزرگی از دیوار پس زمینه در این دو کلاس با کلاس های دیگر تفاوت دارند که احتمالا باعث شده شبکه بتواند این دو کلاس را از بقیه کلاس ها مقدار خوبی جدا کند

اما در مورد بقیه کلاس ها به نظر شبکه اطلاعات مفیدی به ما نمیدهد. در مورد سه کلاس centerlight و normal و glasses no که با طیف های مختلف رنگ قرمز نشان داده شده اند نیز باید گفت که در اکثر نمونه ها به جز نمونه هایی که به طور کلی از عینک استفاده میکنند این سه کلاس بسیار مشابه و حتی در بعضی موارد یکی هستند. شبکه نیز تا حدودی این سه کلاس را نزدیک به هم در نظر گرفته و بیشتر نمونه های این سه کلاس در گوشه پایین و راست جمع شده اند.

در مورد سایر کلاس ها اما به نظر میرسد شبکه اطلاعات مفیدی برای دسته بندی به ما نمیدهد.

۵ دسته بندی تصاویر

در هر دو شبکه ی این بخش از stopping early به عنوان callback استفاده شده. همچنین برای اموزش شبکه SOM فقط از مجموعه اموزش در همه بخش ها استفاده شده و برای کاهش ابعاد مجموعه های اعتبارسنجی و تست نیز از همین شبکه اموزش داده شده روی مجموعه اموزش استفاده میشود.

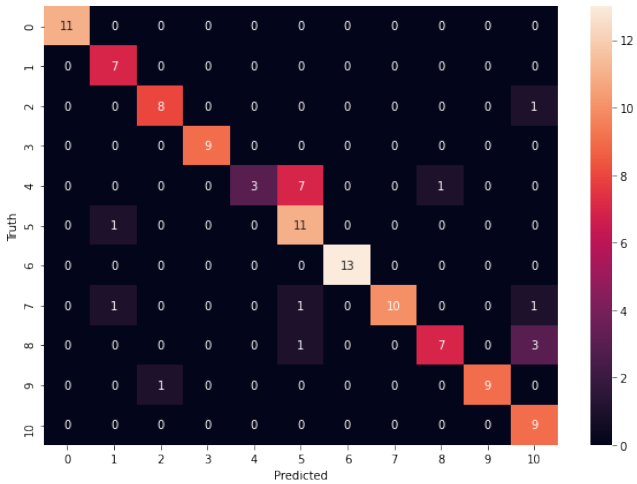
۱.۵ دسته بندی بدون کاهش ابعاد

در این بخش برای دسته بندی تصاویر از یک شبکه با ۳ لایه مخفی استفاده کردم. این مدل به صورت زیر میباشد

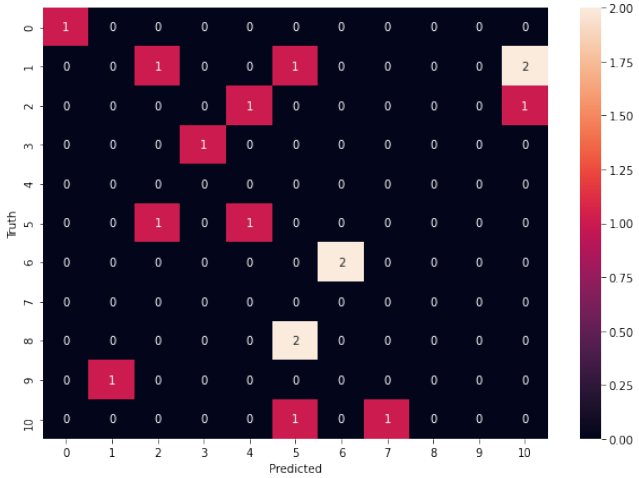
```
model1 = tf.keras.models.Sequential([
                                    layers.Dense(3000, activation='relu' ),
                                    layers.Dense(500, activation='relu'),
                                    layers.Dense(100, activation='relu'),
                                    layers.Dense(11 , activation='softmax')
])
opt = tf.keras.optimizers.Adam(learning_rate=0.0005)
callback = tf.keras.callbacks.EarlyStopping(patience=30)
model1.compile(optimizer=opt,loss ='sparse_categorical_crossentropy' ,metrics=['accuracy'])
```

Listing :۷

این مدل به ۵۱ درصد دقت روی مجموعه اموزش و ۲۴ درصد روی مجموعه اعتبارسنجی و ۲۹ درصد روی مجموعه تست میرسد. همچنین ماتریس در هم ریختگی به صورت زیر هست.



شکل ۳: ماتریس در هم ریختگی برای مجموعه اموزش



شکل ۴: ماتریس در هم ریختگی برای مجموعه تست

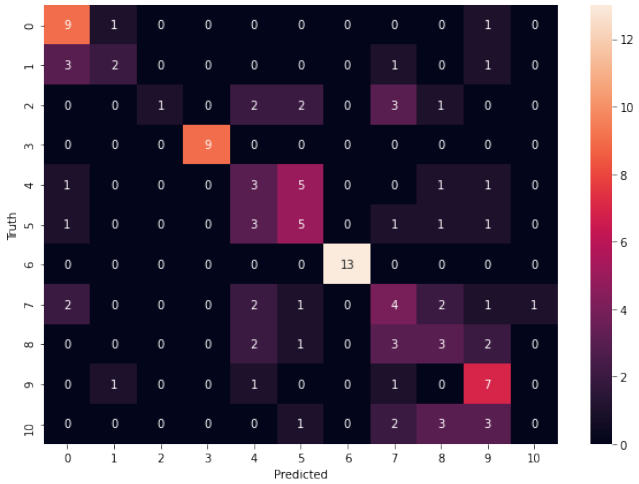
برچسب ها به ترتیب عبارتند از:
'centerlight', 'glasses', 'happy', 'leftlight', 'noglasses', 'normal', 'rightlight', 'sad', 'sleepy', 'surprised', 'wink'

مشاهده میشود در مورد داده های اموزش کلاس noglasses به اشتباه normal دسته بندی شده زیرا در داده ها این دو کلاس گاها یکی هستند. در مورد کلاس های wink و sleepy نیز دسته بند نتوانسته عملکرد خوبی داشته باشد چون شبیه به هم هستند در مورد داده های تست نیز دسته بند عملکرد قابل قبولی نداشته

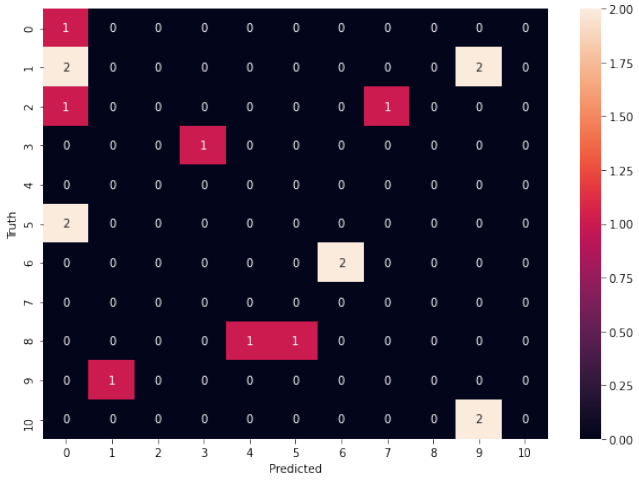
۲.۵ دسته بندی با کاهش ابعاد

از شبکه با مشخصات شبکه بخش قبل نیز در این بخش استفاده میشود فقط برای اموزش از داده هایی که بعد ان ها کاهش یافته استفاده میشود. در این حالت به دقت ۴۷ درصد روی اموزش۲۱ درصد روی اعتبارسنجی و۲۳ درصد روی مجموعه تست میرسیم. ماتریس در هم ریختگی نیز در صفحه بعد امده است. مشاهده میشود در مورد دو کلاس rightlight و leftlight دسته بند عملکرد بهتری داشته نسبت به کلاس های دیگر همانظور که در بخش قبل گفته شد این به این دلیل هست که شبکه SOM توانست این دو کلاس را به خوبی از بقیه کلاس ها جدا کند اما در مورد بقیه کلاس ها چنین نبود همچنین مشاهده میشود که در مورد کلاس های normal و noglasses باز هم بسیار شبیه به هم دسته بند عمل کرده و به نوعی این دو کلاس تفاوتی با هم ندارند.

در کل عملکرد دسته بند با استفاده از کاهش ابعاد بدتر شد. دلیل این موضوع میتواند این باشد که شبکه SOM نتوانست توزیع خوبی در ابعاد پایین تر با توجه به برچسب ها به ما بدهد. و به جز دو کلاس بقیه کلاس ها در ابعاد پایین تر قابل تفکیک نبودند. در واقع ویژگی های جدید که از این روش بدست امده اند مناسب دسته بندی با توجه به این ویژگی ها نبودند.



شکل ۵: ماتریس در هم ریختگی برای مجموعه اموزش



شکل ۶: ماتریس در هم ریختگی برای مجموعه تست