گزارش تمرین سوم هوش محاسباتی

نام تهیه‌کننده: ملیکا نوبختیان

شماره دانشجویی: ۹۷۵۲۲۰۹۴

نسخه: ۱

۱- سوال اول

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	-1	-1
3	1	1	0	0	0	0
4	0	1	0	0	0	0
5	0	-1	0	0	0	-1
6	0	-1	0	0	-1	0

Input pattern = 010000

converged pattern = 111100

threshold = 0

	1	2	3	4	5	6
Input (1=0)	0	1	0	0	0	0
t = 1	1	0	1	1	0	0
t = 2	1	1	1	0	0	0
t = 3	1	1	1	1	0	0
t = 4	1	1	1	1	0	0
$\sum_i x_i v_{ij}$	1	0	1	1	-1	-1
	1	3	1	0	0	0
	2	2	2	1	-1	-1
	2	3	2	1	-1	-1

" $t=3$ و $t=4$ الگویی که به آن همگرا شده است برابر دسادی با 111100 است.

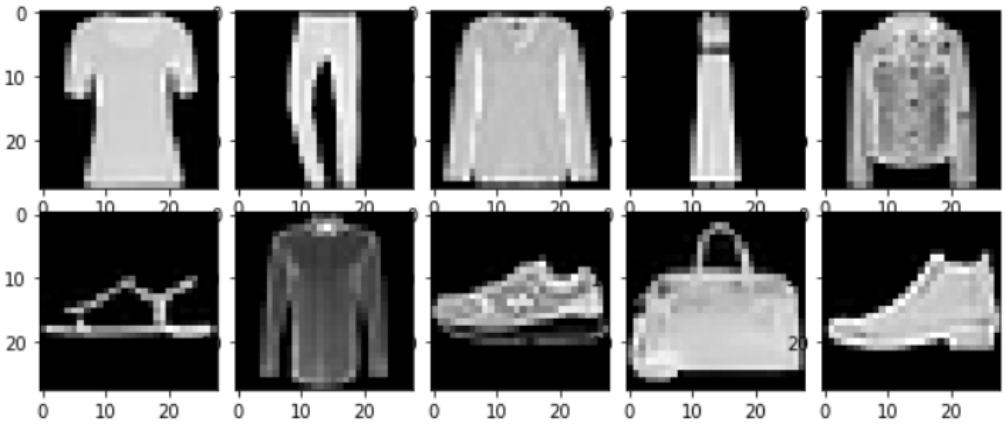
این الگوهمان الگویی است که می خواستیم به آن همگرا شویم پس به پایان کار رسیدیم.

این شبکه با ماتریس وزن داده شده و threshold موجود با شروع از 010000 به 111100 همگرا می شود.

Scanned with CamScanner

۲- سوال دوم

در ابتدا از هر کلاس در دیتاست Fashion MNIST یک تصویر به طور تصادفی انتخاب می‌کنم و به عنوان تصاویر train ذخیره می‌کنم. شکل اولیه این تصاویر (28, 28) است. تصاویر زیر نمونه‌ای از این موارد هستند:



حالا به سراغ تعریف کلاس هاپفیلد می‌رویم. برای ساختن یک نمونه از این کلاس به شکل عکس ورودی نیاز داریم. به این صورت که تعداد نورون‌های شبکه برابر ضرب دو عدد برای شکل عکس ورودی خواهد بود. با توجه به داشتن تعداد نورون‌ها ماتریس وزن هم که مربعی است تشکیل خواهدشد و ابعاد آن همان تعداد نورون‌ها خواهد بود:

```
class Hopfield:

  def __init__(self, input_shape):
    self.num_neurons = input_shape[0] * input_shape[1]
    self.W = np.zeros((self.num_neurons , self.num_neurons))
```

تصاویری که برای یادگیری به هاپفیلد می‌دهیم باید به صورت دودویی یا 1- و ۱ باشند. برای اینکار لازم است یک عملیات پیش پردازش روی تصاویر انجام شود. برای این کار از هر تصویر mean_threshold گرفته می‌شود و براساس آن تعیین می‌شود که چه پیکسل‌هایی ۱ و چه پیکس‌هایی ۱- شوند. در نهایت هم تصویر flat خواهد شد تا برای عملیات محاسباتی بعدی آماده شود:

```
def preprocess_image(self, img):
  img_mean = threshold_mean(img)
  img = np.where(img < img_mean, -1, 1)
  img = img.flatten()

  return img
```

حالا به سراغ پیداکردن وزن‌ها با استفاده از پترن‌های داده‌شده می‌رویم. در ابتدا تمام تصاویر ورودی باید پیش پردازش شوند. در مورد ماتریس وزن‌ها می‌دانیم که قطر آن برابر صفر است و w_ij = w_ji خواهد بود. برای پیداکردن وزن هر عنصر لازم است عنصر i ام در عنصر j ام پترن ضرب شود و این مقدار به خانه ij افزوده شود. هم چنین چون وزن ji برابر ij است این مقدار به ji هم اضافه خواهد شد:

```
def init_weights_with_patterns(self, train_data):
  for data in train_data:
    data = self.preprocess_image(data)
    for i in range(self.num_neurons):
          for j in range(i, self.num_neurons):
              if i != j:
                  w_ij = data[i] * data[j]
                  self.W[i][j] += w_ij
                  self.W[j][i] += w_ij
              else:
                  self.W[i][j] = 0
```

برای اینکه بدانیم شبکه ما هنگام بازیابی یک پترن converge کرده‌است یا نه لازم است انرژی شبکه خود را بدانیم. این مقدار به صورتی که در شکل زیر می‌بینید به دست می‌آید:

```
def energy_function(self, S):
    return -0.5 * np.matmul(np.matmul(S.T,self.W), S)
```

حالا به بخش بازیابی یک پترن می‌رسیم. ورودی این قسمت الگو موردنظر، تعداد iteration ها و هم چنین این انتخاب است که این عمل async باشد یا نه. در ابتدا باید تصویر ورودی پیش پردازش شود. سپس لازم است انرژی اولیه را بر حسب پترن و وزن شبکه محاسبه کنیم. سپس با وزن داخلی وزن در الگو و sign گرفتن از آن پترن جدید را به دست می‌آوریم. حالا باید انرژی جدید را به دست آوریم. اگر انرژی قبلی با جدید یکی باشد یعنی به پایان کار رسیده‌ایم ولی در غیر این صورت به سراغ iteration بعدی می‌رویم و این عمل یا تا پایان iteration ها یا تا رسیدن به انرژی یکسان ادامه خواهد داشت. در صورت async بودن این آپدیت الگو به صورت نورون به نورون انجام خواهد شد:

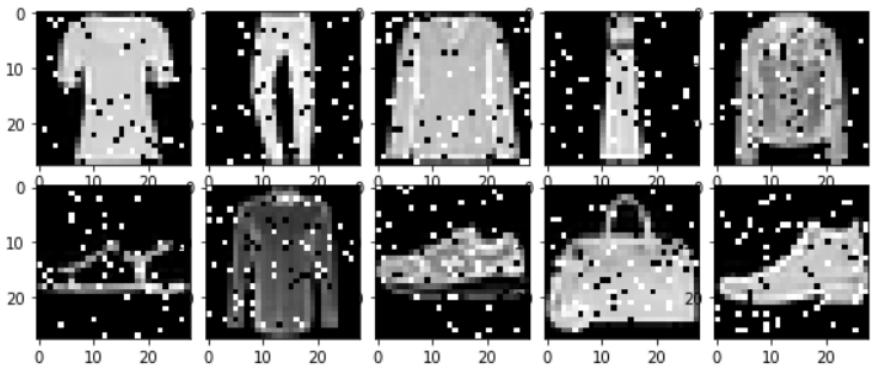
```
def retrieve_pattern(self, pattern, iterations, Async=False, Async_Iter=200):
  new_pattern = self.preprocess_image(pattern)

  if Async == False:

    energy = self.energy_function(new_pattern)

    for i in range(iterations):

      new_pattern = np.sign(np.matmul(self.W, new_pattern))
      new_energy = self.energy_function(new_pattern)

      if energy == new_energy:
        return new_pattern

      energy = new_energy

    return new_pattern
```

```
else:

  energy = self.energy_function(new_pattern)

  for i in range(iterations):
    for j in range(Async_Iter):

      random_neuron = np.random.randint(0, self.num_neurons)
      new_pattern[random_neuron] = np.sign(np.matmul(self.W[random_neuron], new_pattern))

    new_energy = self.energy_function(new_pattern)

    if energy == new_energy:
      return new_pattern

    energy = new_energy

  return new_pattern
```

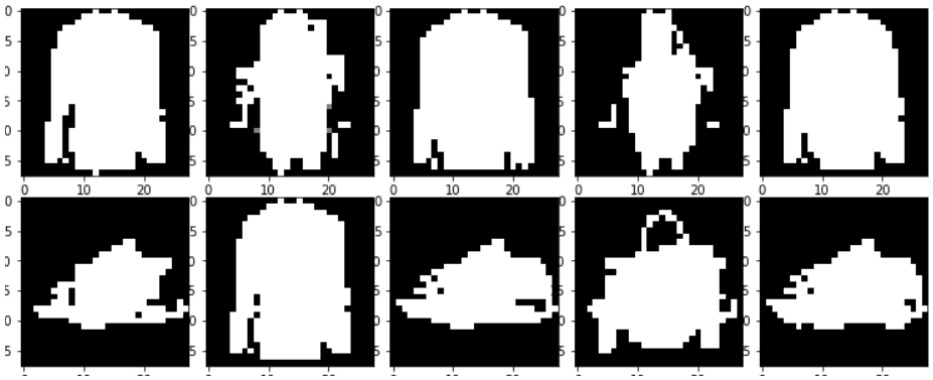
در ابتدا به سراغ ساختن یک هاپفیلد متناسب با ابعاد ورودی (28, 28) یعنی با ۷۸۴ نورون خواهیم رفت و تصاویر مورد نظر را برای آموزش به آن خواهیم داد. پس هر بار با توجه به مقدار موردنظر noise به تصاویر اضافه خواهیم کرد. من برای این کار از salt & pepper استفاده کردم. به صورت زیر من ۱۰ درصد نویز به تصاویر آموزشی اضافه کردم:

```
# add noise
for img in images_to_train:
  noise_img = random_noise(img , mode='s&p',amount=0.1)
  noisy_images.append(noise_img)
```

تصاویر با ۱۰ درصد نویز به شکل زیر خواهند بود:



اگر این تصاویر به شبکه داده شوند نتیجه به صورت زیر خواهد بود:

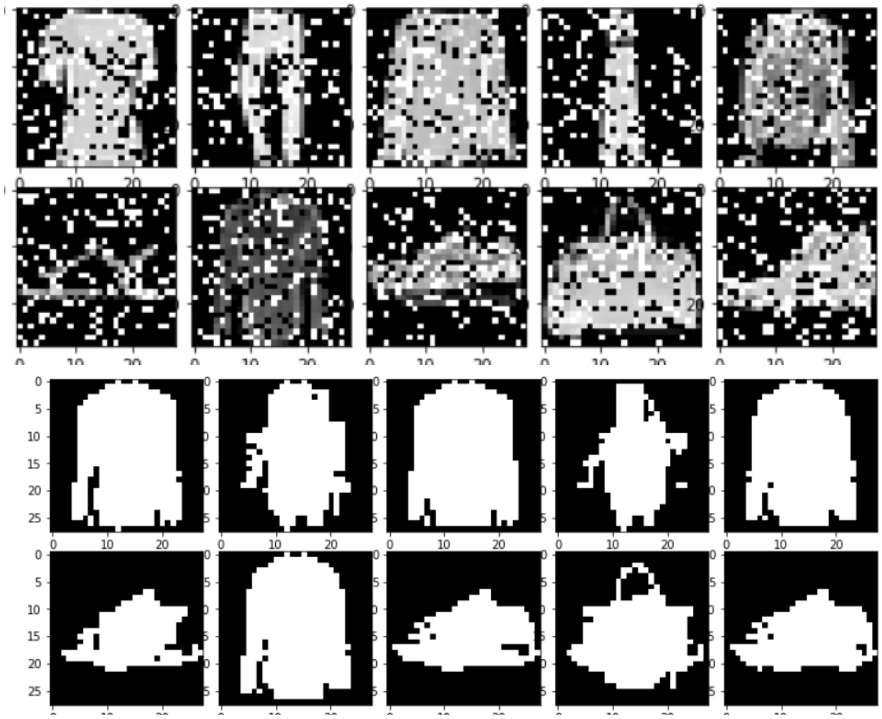


برای محاسبه میزان accuracy برای هر تصویر من به این شکل عمل کردم که تعداد پیکسل‌ها با مقادیر مشابه در تصویر پیش پردازش شده اصلی و تصویر پیش بینی شده را به دست آوردم و نسبت آن به تعداد کل پیکسل‌ها به دست آوردم. سپس از همه مقادیر accuracy میانگین گرفتم و به عنوان دقت کلی ارائه دادم:

```
image 1 accuracy : 0.8303571428571429
image 2 accuracy : 0.8571428571428571
image 3 accuracy : 0.9094387755102041
image 4 accuracy : 0.860969387755102
image 5 accuracy : 0.9323979591836735
image 6 accuracy : 0.8558673469387755
image 7 accuracy : 0.9566326530612245
image 8 accuracy : 0.9209183673469388
image 9 accuracy : 0.8737244897959183
image 10 accuracy : 0.9081632653061225

** Average Accuracy **
Acc: 0.8905612244897959
```

حالا به بررسی مقادیر برای بقیه موارد می پردازیم.

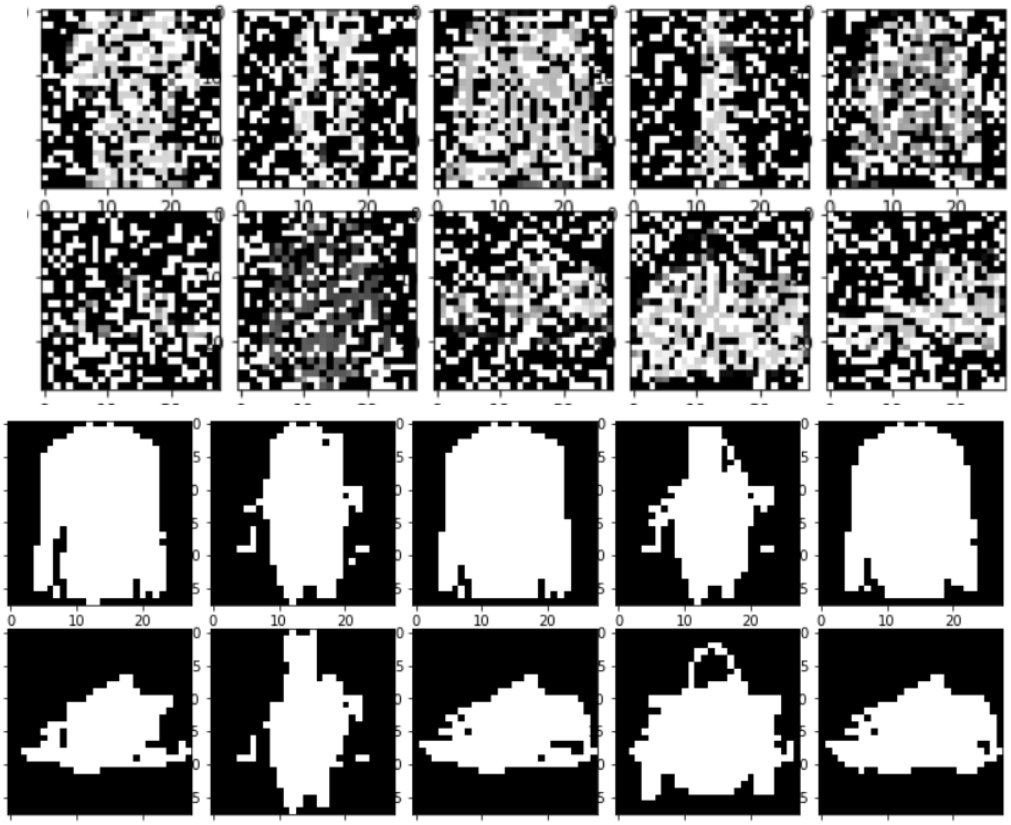
نویز ۳۰ درصد – شبکه ۷۸۴ نورونی



```
image 1 accuracy : 0.8380102040816326
image 2 accuracy : 0.8303571428571429
image 3 accuracy : 0.9094387755102041
image 4 accuracy : 0.8482142857142857
image 5 accuracy : 0.9298469387755102
image 6 accuracy : 0.8558673469387755
image 7 accuracy : 0.9426020408163265
image 8 accuracy : 0.9209183673469388
image 9 accuracy : 0.8571428571428571
image 10 accuracy : 0.9081632653061225

** Average Accuracy **
Acc: 0.8840561224489795
```

دقت کلی : ۸۸ درصد

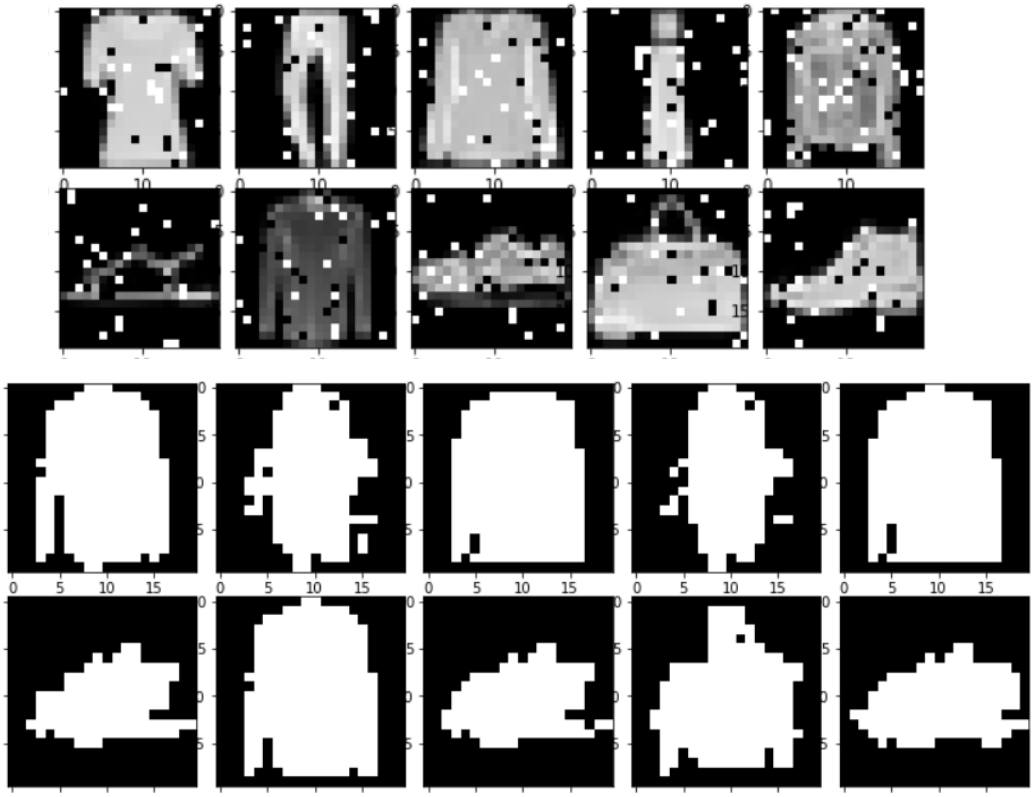
نویز ۶۰ درصد – شبکه ۷۸۴ نورونی



```
image 1 accuracy : 0.8405612244897959
image 2 accuracy : 0.889030612244898
image 3 accuracy : 0.9094387755102041
image 4 accuracy : 0.8443877551020408
image 5 accuracy : 0.9311224489795918
image 6 accuracy : 0.8558673469387755
image 7 accuracy : 0.75
image 8 accuracy : 0.9221938775510204
image 9 accuracy : 0.8852040816326531
image 10 accuracy : 0.9081632653061225

** Average Accuracy **
Acc: 0.8735969387755101
```

دقت کلی: ۸۷ درصد

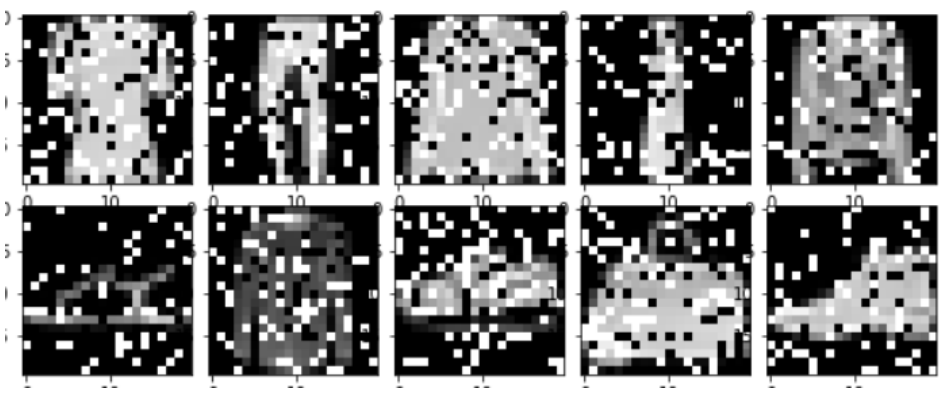
نویز ۱۰ درصد – شبکه ۴۰۰ نورونی

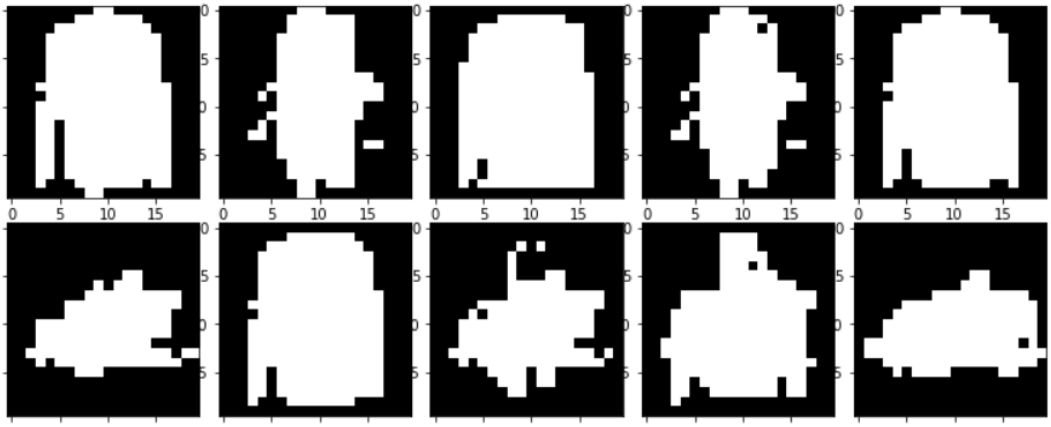


```
image 1 accuracy : 0.8225
image 2 accuracy : 0.8425
image 3 accuracy : 0.94
image 4 accuracy : 0.81
image 5 accuracy : 0.94
image 6 accuracy : 0.865
image 7 accuracy : 0.945
image 8 accuracy : 0.905
image 9 accuracy : 0.8375
image 10 accuracy : 0.895

** Average Accuracy **
Acc: 0.8802500000000002
```

دقت کلی: ۸۸ درصد

نویز ۳۰ درصد – شبکه ۴۰۰ نورونی

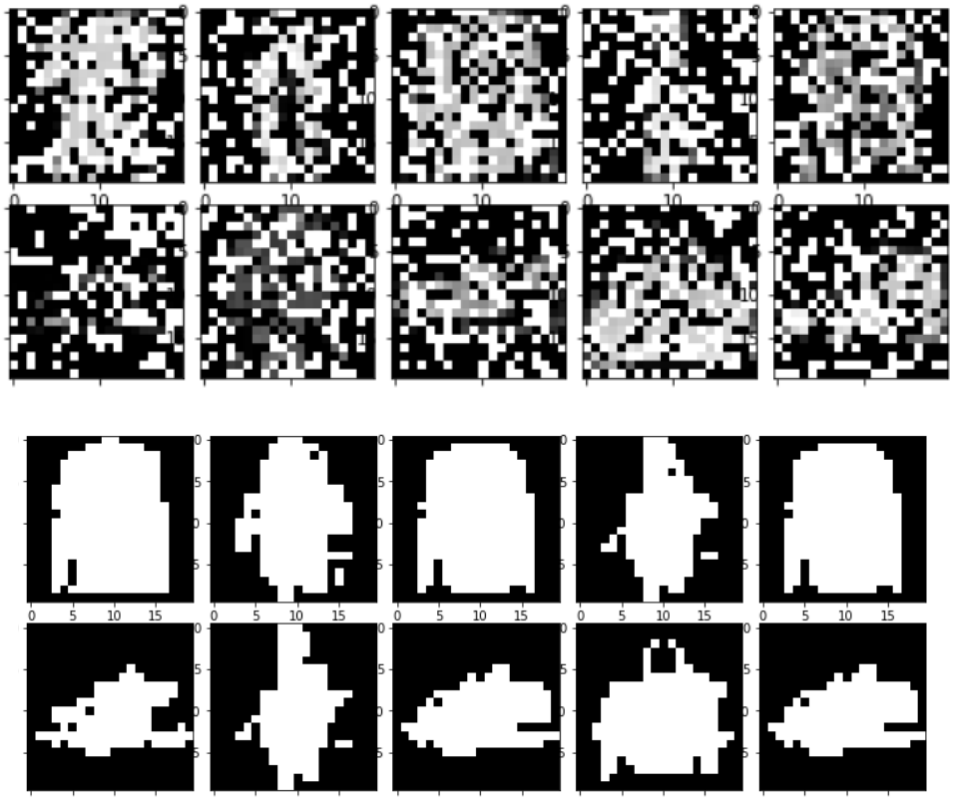




```
image 1 accuracy : 0.8175
image 2 accuracy : 0.8825
image 3 accuracy : 0.94
image 4 accuracy : 0.81
image 5 accuracy : 0.93
image 6 accuracy : 0.8625
image 7 accuracy : 0.9325
image 8 accuracy : 0.8625
image 9 accuracy : 0.8375
image 10 accuracy : 0.905

** Average Accuracy **
Acc: 0.8779999999999999
```

دقت کلی: ۸۷ درصد

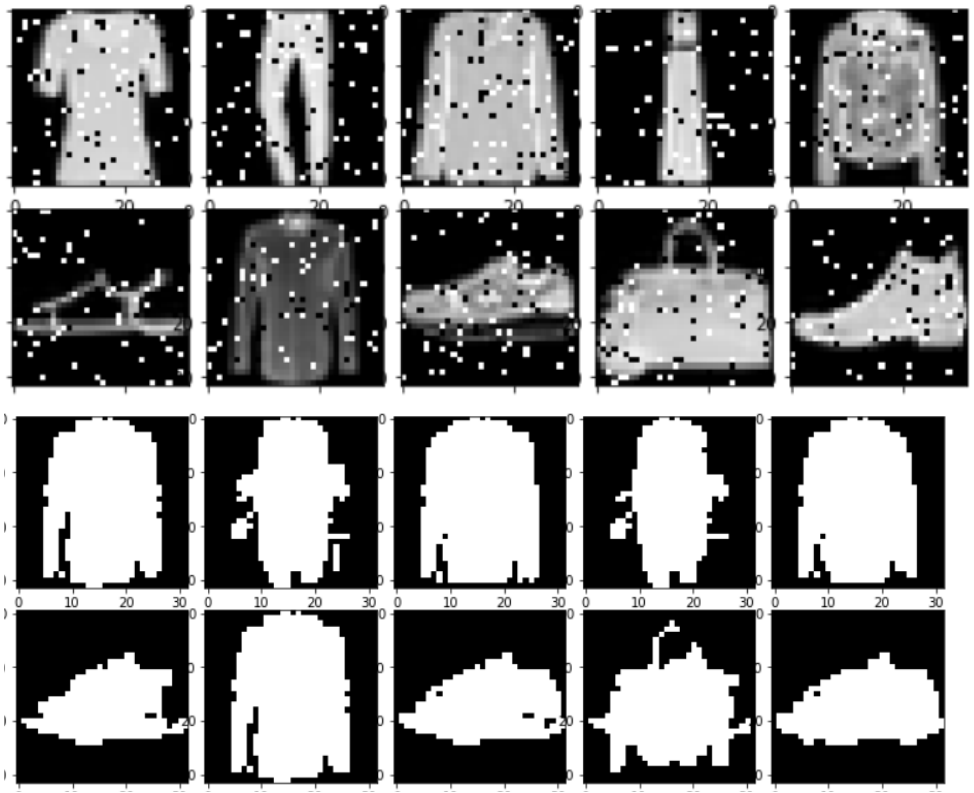
نویز ۶۰ درصد – شبکه ۴۰۰ نورونی



```
image 1 accuracy : 0.82
image 2 accuracy : 0.83
image 3 accuracy : 0.9225
image 4 accuracy : 0.85
image 5 accuracy : 0.9225
image 6 accuracy : 0.9025
image 7 accuracy : 0.7575
image 8 accuracy : 0.92
image 9 accuracy : 0.8775
image 10 accuracy : 0.8825

** Average Accuracy **
Acc: 0.8685
```

دقت کلی: ۸۶ درصد

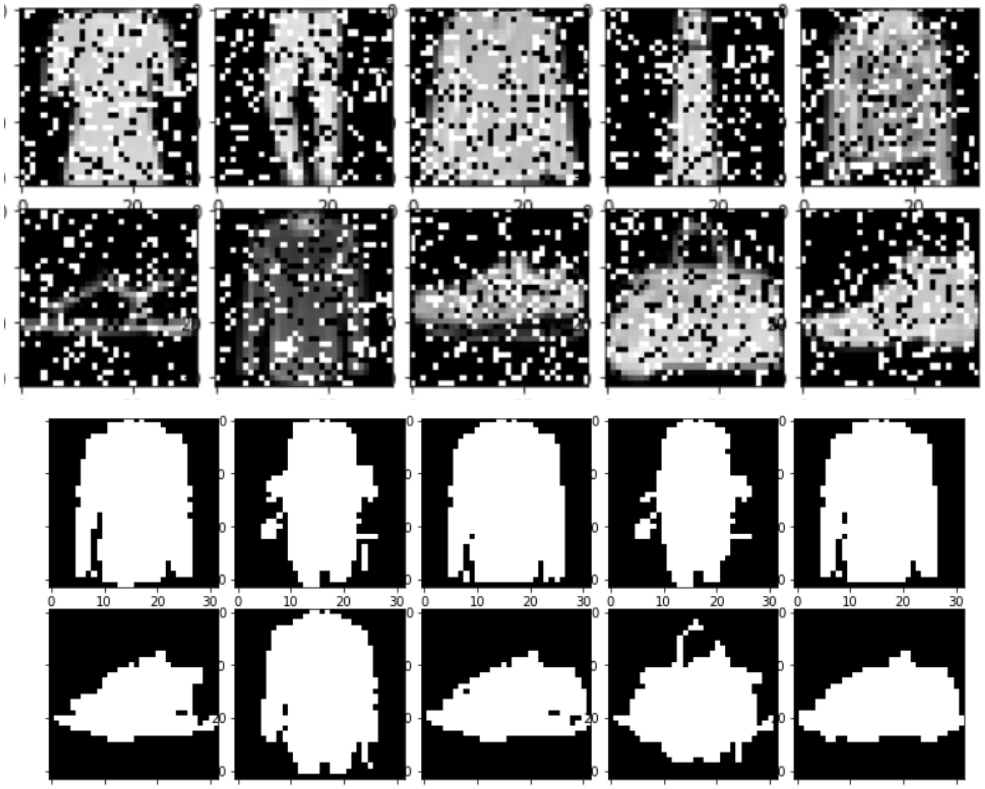
نویز ۱۰ درصد – شبکه ۱۰۲۴ نورونی



```
image 1 accuracy : 0.83984375
image 2 accuracy : 0.8701171875
image 3 accuracy : 0.912109375
image 4 accuracy : 0.8154296875
image 5 accuracy : 0.9287109375
image 6 accuracy : 0.8681640625
image 7 accuracy : 0.96484375
image 8 accuracy : 0.9306640625
image 9 accuracy : 0.8671875
image 10 accuracy : 0.9130859375

** Average Accuracy **
Acc: 0.891015625
```

دقت کلی: ۸۹ درصد

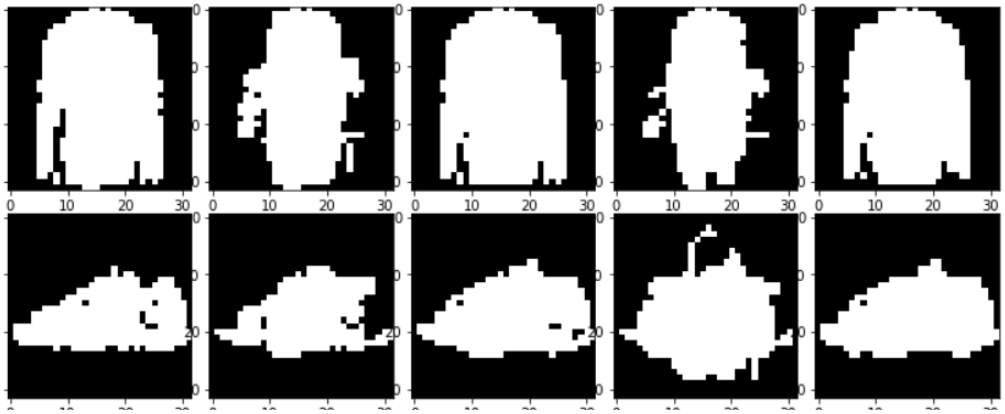
نویز ۳۰ درصد – شبکه ۱۰۲۴ نورونی



```
image 1 accuracy : 0.8388671875
image 2 accuracy : 0.8701171875
image 3 accuracy : 0.912109375
image 4 accuracy : 0.8154296875
image 5 accuracy : 0.921875
image 6 accuracy : 0.8681640625
image 7 accuracy : 0.9326171875
image 8 accuracy : 0.9306640625
image 9 accuracy : 0.8486328125
image 10 accuracy : 0.912109375

** Average Accuracy **
Acc: 0.88505859375
```

دقت کلی: ۸۸ درصد

نویز ۶۰ درصد – شبکه ۱۰۲۴ نورونی



```
image 1 accuracy : 0.845703125
image 2 accuracy : 0.8515625
image 3 accuracy : 0.912109375
image 4 accuracy : 0.8154296875
image 5 accuracy : 0.9287109375
image 6 accuracy : 0.880859375
image 7 accuracy : 0.646484375
image 8 accuracy : 0.9287109375
image 9 accuracy : 0.8486328125
image 10 accuracy : 0.9130859375

** Average Accuracy **
Acc: 0.85712890625
```

دقت کلی: ۸۵ درصد

	۷۸۴	۴۰۰	۱۰۲۴
10%	89.0	88.0	89.1
30%	88.4	87.7	88.5
60%	87.3	86.8	85.7

۳- سوال سوم

۳-۱ Hopfield Network

TSP با استفاده از Hopfield قابل حل است. اگر تعداد شهرهایی که بخواهیم بین آن‌ها یک مسیر پیدا کنیم n باشد، لازم است یک شبکه هاپفیلد متناظر با آن با تعداد n نورون ایجاد کنیم. برای اینکه وزن های این شبکه را init کنیم از فاصله بین خود شهرها استفاده می کنیم. حالا شروع به آموزش شبکه می‌کنیم. هر وزنی که یک بشود نشان دهنده جایگاه آن شهر در مسیر خواهد بود. هر زمان که هر کدام از نورون های شبکه در جایگاه یک نورون متفاوت یک شود به مسیر خود رسیده ایم و این مسیر بهترین مسیر ما خواهد بود.

۳-۲ MLP

این مسئله با استفاده از mlp قابل حل نیست. برای حل کردن این مسئله باید یک label را از قبل برای هر شهر بدانیم ولی در این مسئله ما اطلاعی از جایگاه شهرها در مسیر نداریم پس عملا هیچ چیزی نداریم که با آن مسئله را به صورت classification با استفاده از mlp حل کنیم.

SOM -۳-۳

این مسئله با استفاده از som قابل حل است. برای این کار یک شبکه som با تعداد n نورون که همان تعداد شهرها است می‌سازیم. بردار وزن هر نورون ۲ عنصر خواهد داشت. ابتدا وزن‌های شبکه را به صورت رندوم init می کنیم. ورودی های شبکه مختصات شهرها خواهند بود که بهتر است آن ها را نرمال کنیم. هر بار یک نورون برنده خواهد شد و در نهایت این شبکه شکل خواهد گرفت و مسیر ها مشخص و برای هر شهر تنها یک نورون برنده خواهیم داشت که جایگاه آن شهر را در مسیر مشخص خواهد کرد.

RBF -۳-۴

منبع:
https://www.researchgate.net/publication/279200277_Solving_Traveling_Salesman_Problem_in_Radial_Basis_Function_Network

این مسئله را می‌توانیم به کمک RBF حل کنیم. ابتدا باید min فاصله میان دو شهر و max فاصله میان آن‌ها را پیدا کنیم که بدانیم توری که در نهایت میان شهرها پیدا می‌کنیم، حداقل و حداکثر طول آن چقدر خواهد بود. حالا فاصله ای که داریم را به بازه‌های satisfactory تقسیم می‌کنیم و مرکز آن را مشخص می‌کنیم و همین طور width آن را. حالا در هر interval می‌توانیم تورهای satisfactory که کمترین فاصله میان شهرها را دارند پیدا کنیم و در نهایت به مسیر موردنظرمان برسیم.

۵-۳- حل مسئله با استفاده از SOM

ابتدا باید مختصات شهرها را بخوانیم و مختصات x و y و تمام اطلاعات به طور کلی را ذخیره می‌کنیم:

```
file = open('Cities.csv')
csvreader = csv.reader(file)
rows = []
x = []
y = []
for row in csvreader:
        x_point = float(row[0].split()[1])
        y_point = float(row[0].split()[2])
        x.append(x_point)
        y.append(y_point)
        rows.append([x_point, y_point ])
coordinates = np.array(rows)
```

نرمال سازی مقادیری که برای مختصات داریم می‌تواند به ما کمک کند. برای همین نسبت قطری را با توجه به max و min به دست آورده و با کم کردن مقدار min هر قسمت از آن تقسیم بر نسبت موردنظر آن‌ها را نرمال می‌کنیم:

```
ratio = np.sqrt((max(x) - min(x)) * (max(x) - min(x)) + (max(y) * min(y)) * (max(y) * min(y)))
normal_coordinates =  ( coordinates - np.array([min(x), min(y)]))/ ratio
```

در ادامه به توضیح چند تابع مفید می‌پردازیم:

```
def som_network(size):
    return np.random.rand(size, 2)
```

این تابع شبکه som ما را خواهد ساخت که با گرفتن تعداد شهرها شبکه ای به این ابعاد خواهد ساخت.

```
def closest_neuron(network, city):
  dist = network - city
  dist = dist ** 2
  dist = np.sum(dist, axis=1)
  return np.where(dist == np.amin(dist))
```

این تابع با توجه به شبکه‌ای که داریم و مختصات شهری که انتخاب شده‌است نورون برنده را حساب خواهد کرد.

```
def get_neighborhood(center, radix, domain):
    if radix < 1:
      radix = 1
    deltas = np.absolute(center - np.arange(domain))
    distances = np.minimum(deltas, domain - deltas)

    return np.exp(-(distances*distances) / (2*(radix*radix)))
```

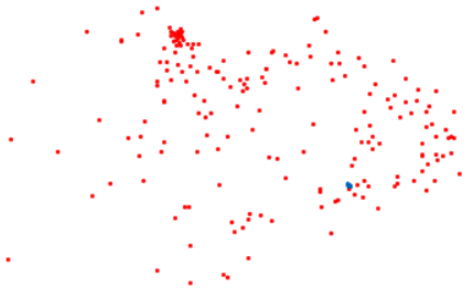
تابع بالا نیز مقادیر همسایگی را حساب می‌کند. Center همان نورون برنده خواهد بود و radix همان شعاع خواهد بود که ما نسبتی از تعداد شهرها را همیشه به عنوان شعاع در نظر خواهیم گرفت. در آخر domain هم همان تعداد شهرها را مشخص می‌کند. در نهایت تابع گاوسی محاسبه می‌شود و مقادیر همسایگی محاسبه می‌شوند.

```
def plot_city_network(network, coordinates):
  fig = plt.figure(figsize=(5, 5), frameon = False)
  axis = fig.add_axes([0,0,1,1])

  axis.set_aspect('equal', adjustable='datalim')
  plt.axis('off')

  axis.scatter(coordinates[:, 0], coordinates[:, 1], color='red', s=4)
  axis.plot(network[:,0], network[:,1], 'r.', ls='-', color='#0063ba', markersize=2)

  plt.show()
```

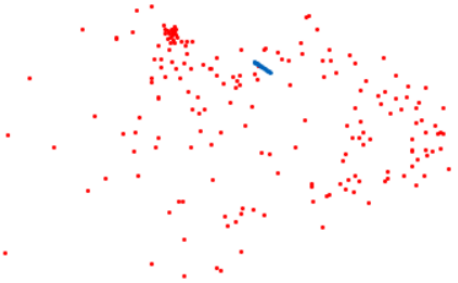
تابع بالا هم برای ما مختصات نقاط شهرها را رسم می‌کند و هر بار با توجه به وزن‌های شبکه تور بین شهرها را رسم می‌کند.

حالا نوبت به آموزش شبکه می‌رسد. پس از هر ۲۰۰۰ آپدیت تصویر شبکه رسم خواهد شد که به شکل زیر است:

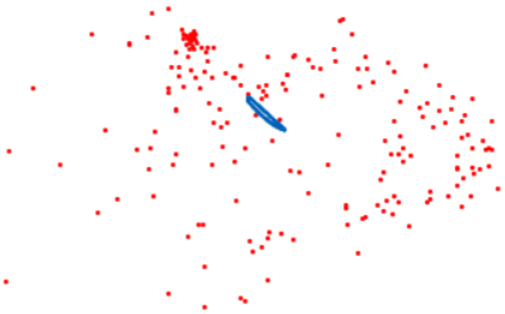
epoch 2000 : Cities and Network



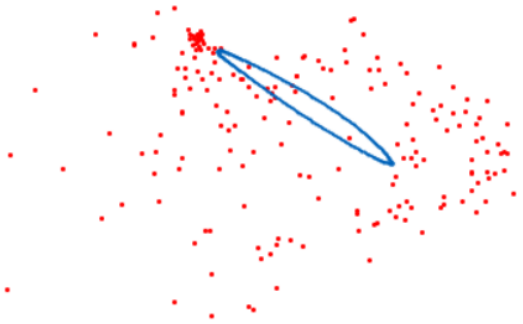
epoch 4000 : Cities and Network



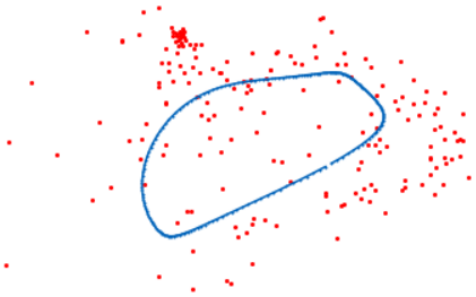
epoch 6000 : Cities and Network



epoch 8000 : Cities and Network

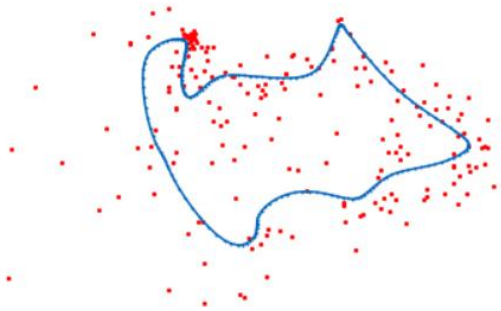


epoch 10000 : Cities and Network

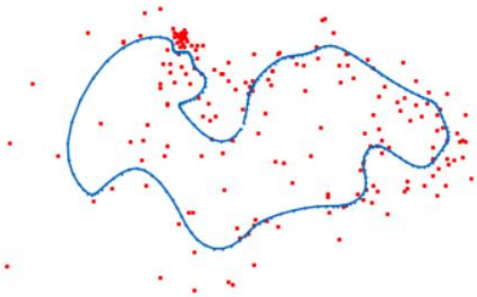


epoch 12000 : Cities and Network

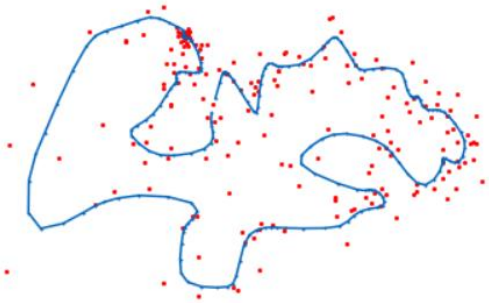
epoch 14000 : Cities and Network



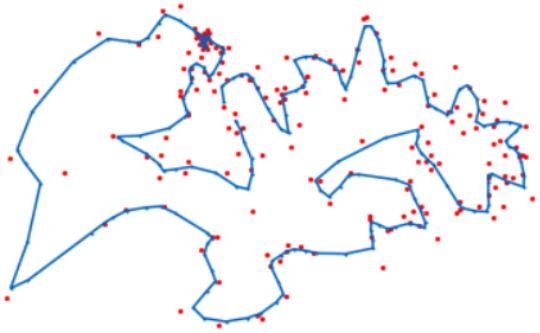
epoch 16000 : Cities and Network



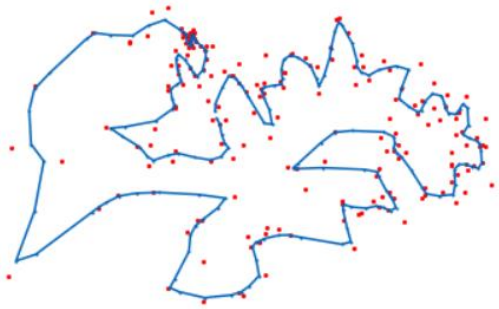
epoch 18000 : Cities and Network



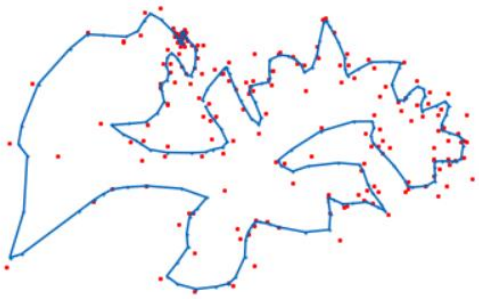
epoch 20000 : Cities and Network



epoch 22000 : Cities and Network



epoch 24000 : Cities and Network



تصاویر زیر نمونه‌هایی از نتیجه نورون برنده برای شهرهای متفاوت را نشان می‌دهد. در نهایت این مسئله به converge نهایی نرسیده‌است ولی قسمتی از نتایج به شکل زیر است:

```
city 1 ==> (x, y) : (0.0, 2.7984115875337372e-08), location in tour : 134
city 2 ==> (x, y) : (4.028035470022631e-09, 1.6991875114721342e-07), location in tour : 131
city 3 ==> (x, y) : (3.0104142577492063e-08, 2.3966708014153446e-07), location in tour : 129
city 4 ==> (x, y) : (5.957226951230243e-08, 1.5582070332248925e-07), location in tour : 132
city 5 ==> (x, y) : (9.455244297660563e-08, 2.9849736320418830e-07), location in tour : 127
city 6 ==> (x, y) : (9.98524715718145e-08, 1.0409244865576955e-07), location in tour : 136
city 7 ==> (x, y) : (1.0780249538453852e-07, 1.9408688764699208e-07), location in tour : 143
city 8 ==> (x, y) : (1.2190058136943847e-07, 1.1861453234906445e-07), location in tour : 137
city 9 ==> (x, y) : (1.3419662939343738e-07, 2.8620131518019076e-07), location in tour : 126
city 10 ==> (x, y) : (1.3546862251862362e-07, 2.881093239480587e-07), location in tour : 126
city 11 ==> (x, y) : (1.4257065091455774e-07, 1.7235676659061899e-07), location in tour : 142
city 12 ==> (x, y) : (1.5359470581336903e-07, 2.9425734796005885e-07), location in tour : 124
city 13 ==> (x, y) : (1.5645671896517095e-07, 1.5147467911121574e-07), location in tour : 140
city 14 ==> (x, y) : (1.5794073002464395e-07, 1.7331077097455296e-07), location in tour : 141
city 15 ==> (x, y) : (1.6006075672679724e-07, 3.2065144380878056e-07), location in tour : 122
city 16 ==> (x, y) : (1.6048475443519357e-07, 1.2200655217640399e-07), location in tour : 139
```