گزارش تمرین دوم هوش محاسباتی

نام تهیه‌کننده: ملیکا نوبختیان

شماره دانشجویی: ۹۷۵۲۲۰۹۴

نسخه: ۱

۱- شبکه Kohonen

در ابتدا با استفاده از keras دیتاست MNIST را لود کردم. سپس به سراغ انتخاب ۵۰۰۰ نمونه از MNIST رفتم. برای اینکه یک دیتاست یکنواخت از نقاط داشته باشم از هر کلاس از نقاط یعنی اعداد ۰ تا ۹ به تعداد ۵۰۰ نمونه انتخاب کردم و به این ترتیب دیتاست کلی ۵۰۰۰ تایی را ساختم:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

## Create a balanced dataset from MNIST - We have 500 samples from each number

dataset_count = 0
idx = 0
zero = 0; one = 0; two = 0; three = 0; four = 0; five = 0; six = 0; seven = 0; eight = 0; nine = 0
dataset_x = []
dataset_y = []
while dataset_count != 5000:
  if y_train[idx] == 0 and zero < 500:
    dataset_x.append(x_train[idx])
    dataset_y.append(0)
    dataset_count +=1
    zero += 1
    idx +=1
    continue
```

```
if y_train[idx] == 2 and two < 500:
  dataset_x.append(x_train[idx])
  dataset_y.append(2)
  dataset_count +=1
  two += 1
  idx +=1
  continue


if y_train[idx] == 3 and three < 500:
  dataset_x.append(x_train[idx])
  dataset_y.append(3)
  dataset_count +=1
  three += 1
  idx +=1
  continue


if y_train[idx] == 4 and four < 500:
  dataset_x.append(x_train[idx])
  dataset_y.append(4)
  dataset_count +=1
  four += 1
  idx +=1
  continue
```

این منوال برای اعداد دیگر نیز ادامه می‌یابد و در نهایت dataset_x و dataset_y مجموعه آموزش ما را تشکیل خواهند داد. پس از این مرحله لازم است با ساختار کلی kohonen آشنا شویم.

برای initialize اولیه یک kohonen map لازم است ابتدا ابعاد آن را بدانیم. x و y طول و عرض این map خواهند بود. هم چنین لازم است ابعاد ورودی را نیز بدانیم زیرا با آن است که می‌توانیم ابعاد وزن نورون‌های شبکه را مشخص کنیم. نرخ یادگیری و شعاع همسایگی نیز از مواردی که لازم است برای initialize کردن بدانیم:

```
class KohonenMap:

  def __init__(
      self,
      x,
      y,
      input_size,
      neighbourhood_radius,
      lr
  ):
    self.x = x
    self.y = y
    self.input_size = input_size
    self.radius = neighbourhood_radius
    self.lr = lr

    ## build map for computing neighbourhood
    self.X_map , self.Y_map = self.build_map_index()

    ## init weights
    self.weights = self.init_weights()
```

می‌دانیم در مواقعی که می‌خواهیم فاصله یک نورون را از نورون‌های دیگر بدانیم لازم است که ایندکس‌های آن نقاط را هر بار منهای نقطه موردنظر کنیم. اما هر بار به دست آوردن مختصات نقطه‌ها در هر بار محاسبه فاصله سرعت کار ما را کند خواهد کرد. به همین منظور تابع build_map_index به کمک ما می‌آید و دو map مجزا از مختصات x وy نقاط موجود در map به ما می‌دهد تا محاسبات ما را بهینه‌تر و پرسرعت‌تر کند:

```
def build_map_index(self):

  x_range = np.arange(self.x)
  y_range = np.arange(self.y)

  map_x, map_y = np.meshgrid(x_range, y_range)

  return map_x, map_y
```

برای init کردن وزن‌ها هم تابع init_weights به کمک می‌آید و با توجه به ابعاد شبکه و هم چنین ابعاد داده ورودی وزن‌ها را init می‌کند:

```
def init_weights(self):

  weight_map = np.zeros((self.x, self.y, self.input_size[0], self.input_size[1]))
  for i in range(self.x):
    for j in range(self.y):
      neuron_weight = np.random.randint(0, 256,(self.input_size[0],self.input_size[1]), dtype='int64')
      weight_map[i][j] = neuron_weight

  return weight_map
```

تابع گاوسی یکی از مهم‌ترین توابع در kohonen است. این تابع براساس فاصله تمام نقاط شبکه از نقطه برنده که برای ما حکم مرکز دارد و هم چنین مقدار شعاع یا همسایگی که در ابتدا تعیین کردیم، میزانی که وزن‌های هر نورون تغییر خواهدکرد را نشان خواهد داد:

```
def gaussian_function(
    self,
    center,
    radius
):
  neighbour = 2 * radius * radius
  h_map = np.zeros((self.x, self.y))

  x_dist = (self.X_map - center[0]) ** 2
  y_dist = (self.Y_map - center[1]) ** 2
  dist_all = np.sqrt(x_dist + y_dist)

  h_map = np.exp(-dist_all / neighbour).T

  ## extend h map
  extended_h_map = np.zeros((self.x, self.y, self.input_size[0], self.input_size[1]))
  for i in range(self.x):
    for j in range(self.y):
      element = np.full((28, 28), h_map[i][j])
      extended_h_map[i][j] = element


  return extended_h_map
```

یک تابع دیگر در اینجا winner_neuron است که نورون برنده در شبکه براساس وزن‌های آن و داده‌ای که به ما داده می‌شود، مشخص خواهد کرد. معیار برنده‌بودن کمترین فاصله اقلیدسی بین نقطه موردنظر و وزن‌های نورون‌های مختلف شبکه است:

```
def winner_neuron(
    self,
    data
):
  distance = (self.weights - data) ** 2
  distance = np.sum(distance, axis=(2,3))
  distance = np.sqrt(distance)

  return np.where(distance == np.amin(distance))
```

بسته به اینکه می‌خواهیم از شعاع ثابت یا متغیر در طول یادگیری شبکه استفاده کنیم، تابع زیر به کمک ما خواهد آمد. در صورتی که بخواهیم از شعاع متغیر استفاده کنیم که شعاع هر بار تقسیم بر یک به علاوه شماره‌ای از داده که درآن هستیم تقسیم بر کل تعداد داده‌ها خواهد شد:

```
def compute_new_radius(
    self,
    data_len,
    iteration,
    constant_radius
):
  if constant_radius == True:
    return self.radius
  else:
    return self.radius / (1 + iteration/data_len)
```

از تابع plot_figures هم برای کشیدن وزن‌های شبکه در هر مرحله استفاده می‌کنیم تا ببینیم هر بار شبکه در چه حالتی قرار دارد و چقدر به حالت دلخواه ما نزدیک شده‌است. در واقع وزن هر نورون در اینجا یک تصویر سیاه و سفید است که در نهایت به شکل‌های نهایی ما نزدیک خواهد شد:

```
def plot_figures(self, figures, nrows, ncols):
  fig, axeslist = plt.subplots(ncols=ncols, nrows=nrows, figsize=(7,7))
  for i in range(nrows):
    for j in range(ncols):
      axeslist.ravel()[i*nrows + j].imshow(figures[i][j], cmap='gray')
      axeslist.ravel()[i*nrows + j].set_axis_off()

  plt.show()
```

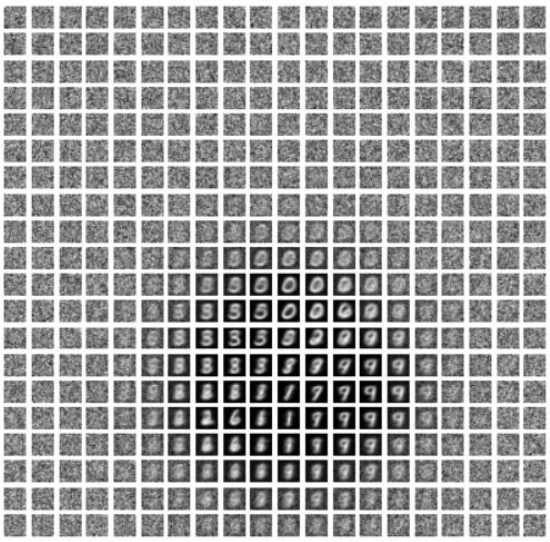
تابع train به صورت زیر است:

```
def train(
    self,
    X_data,
    Y_data,
    epochs,
    constant_radius=False
):
  data_count = X_data.shape[0]
  for e in range(epochs):
    np.random.shuffle(X_data)
    print(f'epoch {e+1}/{epochs}:')
    for i in range(data_count):
      winner_pos = self.winner_neuron(X_data[i])
      winner_pos = (winner_pos[0][0], winner_pos[1][0])
      if (i + 1) % 500 == 0:
        print(f"   example {i+1}/{data_count} ===> winner neuron : ({winner_pos[0]}, {winner_pos[1]}) , actual label : {Y_data[i]} ")
      current_radius = self.compute_new_radius(data_count, i, constant_radius)
      h_map = self.gaussian_function(winner_pos, current_radius)
      subb = self.weights - X_data[i]
      delta_w = self.lr * h_map * -( self.weights - X_data[i]  )
      self.weights = self.weights + delta_w
    self.plot_figures(self.weights, self.x, self.y)
```

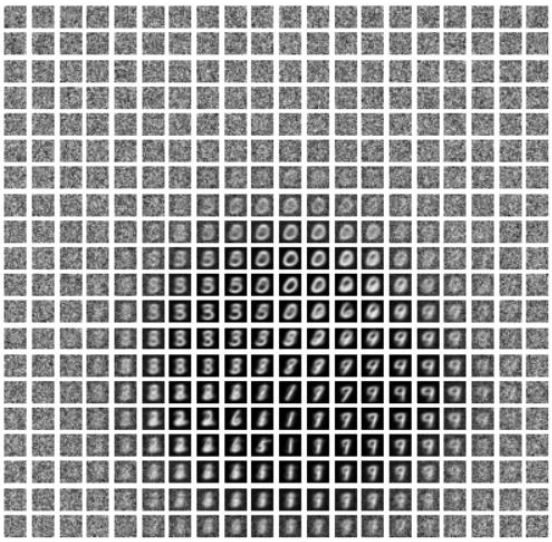
ورودی‌های این تابع داده‌های ورودی‌ها X_data، برچسب اصلی داده‌های ورودی Y_data، تعداد epochs و در نهایت تصمیم به اینکه آیا از شعاع ثابت استفاده کنیم یا نه خواهد بود. در هر epoch برای اینکه توزیع داده‌ها همیشه یکنواخت نباشد، داده‌ها را shuffle خواهیم کرد. ابتدا در برای هر داده ورودی نورون برنده را مشخص خواهیم کرد. سپس شعاع همسایگی را محاسبه خواهیم کرد که بسته به ثابت‌بودن یا نبودن آن نتیجه تفاوت خواهد کرد. حالا با وجود نورون برنده و شعاع همسایگی، مقدار تابع گاوسی را برای تمام نقاط شبکه حساب کنیم. حالا با داشتن این موارد می‌توانیم تغییرات وزن را محاسبه کنیم. همان طور که در تصویر مشخص است مقادیر همسایگی در نرخ یادگیری و در تفاوت وزن‌های شبکه و نقطه داده‌شده مقدار تغییرات وزن را به ما خواهند داد. در نهایت پس از اتمام هر epoch وزن‌های شبکه را ترسیم خواهیم کرد.

۱-۱- ضریب یادگیری

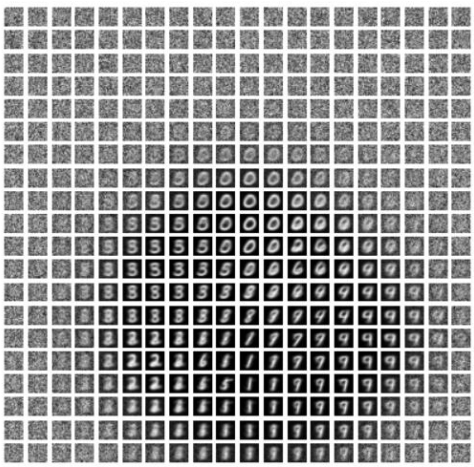
برای آزمایش تاثیر ضریب یادگیری بر آموزش یک kohonen ابتدا سعی کردم شبکه 20*20 را نرخ یادگیری 0.01 و هم چنین شعاع همسایگی متغیر آزمایش کنم. برای این کار 10 epoch به آن را بررسی کردم. شکل وزن‌ها در هر مرحله به صورت زیر است:



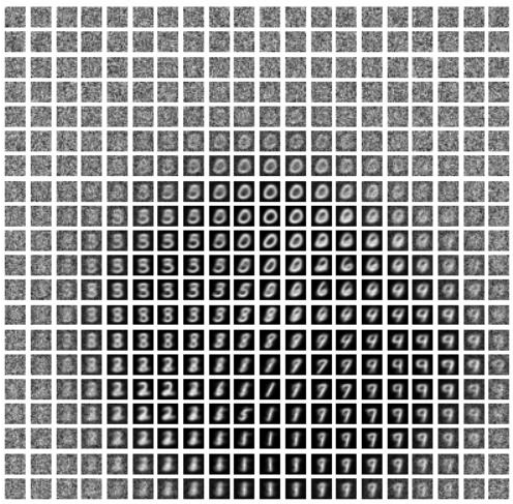
شکل ۳- epoch 1



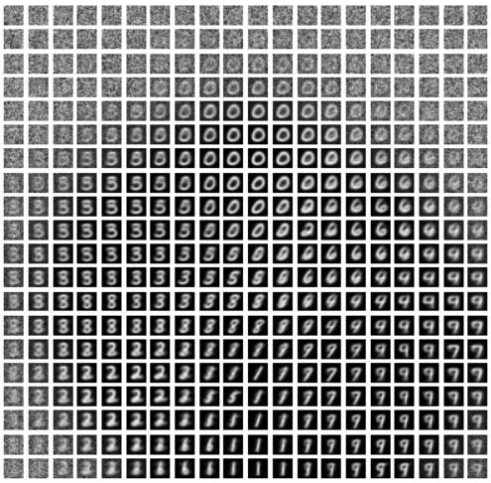
شکل ۲- epoch 2



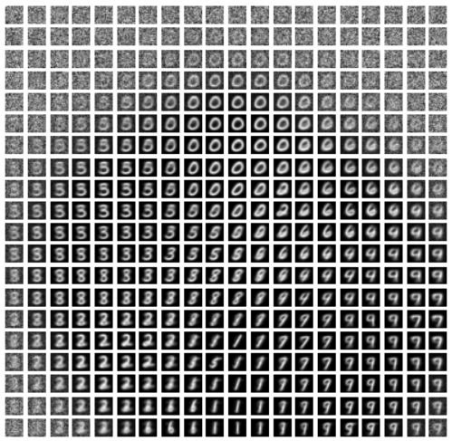
شکل ۱- epoch 3



شکل ۶- epoch 4



شکل ۴- epoch 9

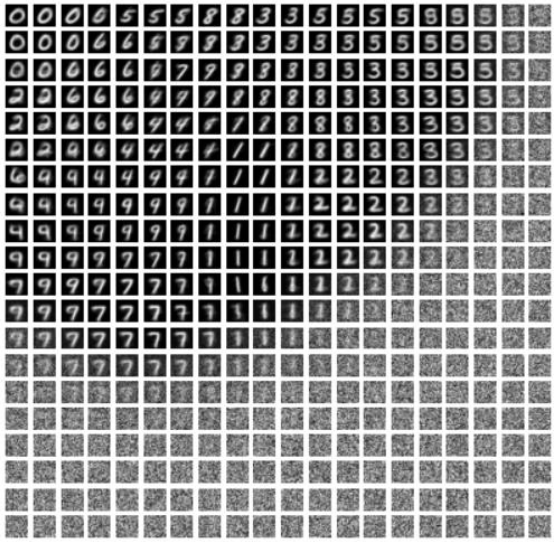


شکل ۵- epoch 10

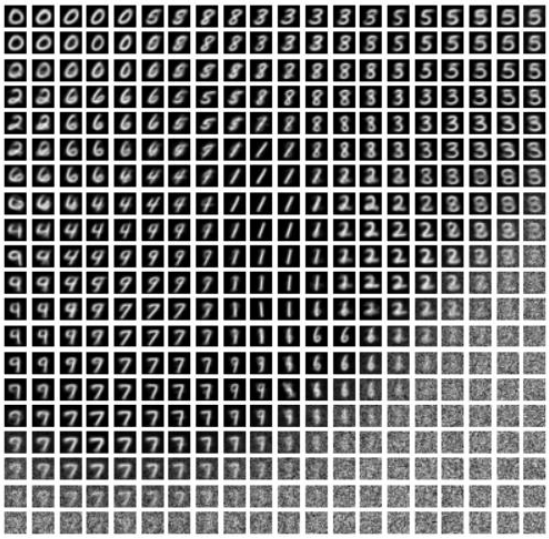
همان طور که در تصاویر مشاهده می‌کنید با این نرخ یادگیری سرعت تغییر شبکه بسیار کم است و زمان زیادی طول می‌کشد تا شبکه بتواند خروجی‌های مطلوب را یاد بگیرد و به نمایش مناسبی از آن‌ها برسد. در این حالت

هنوز شبکه ما به حالت کامل خود نرسیده‌است و ایده‌آل و مناسب نیست.

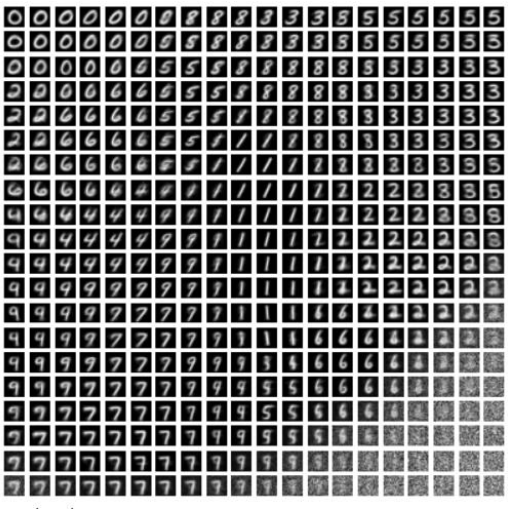
این بار سعی می‌کنیم همان شبکه را با نرخ یادگیری بالاتری امتحان کنیم. این بار نرخ یادگیری را برابر 0.1 قرار می‌دهیم که ۱۰ برابر حالت قبل خواهد بود. وزن‌ها شبکه در epoch های مختلف به شکل‌های زیر خواهد بود:



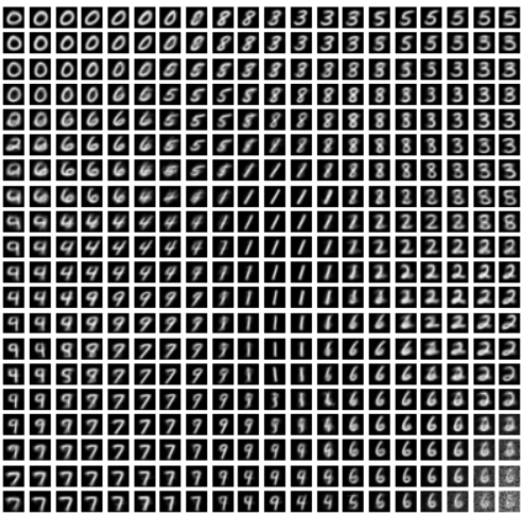
شکل ۷- epoch 1



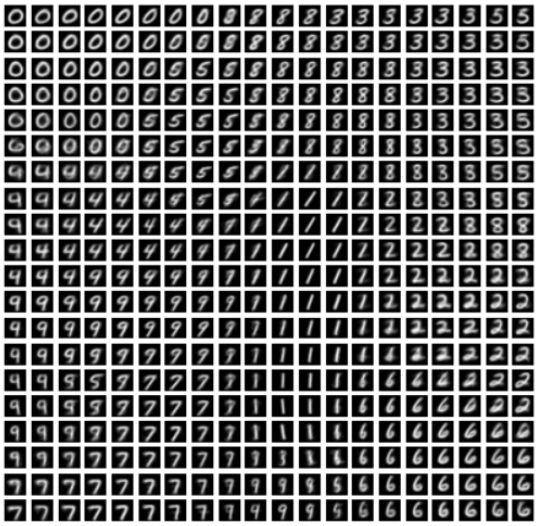
شکل ۸- epoch 2



شکل ۱۱- epoch 3



شکل ۱۰- epoch 5

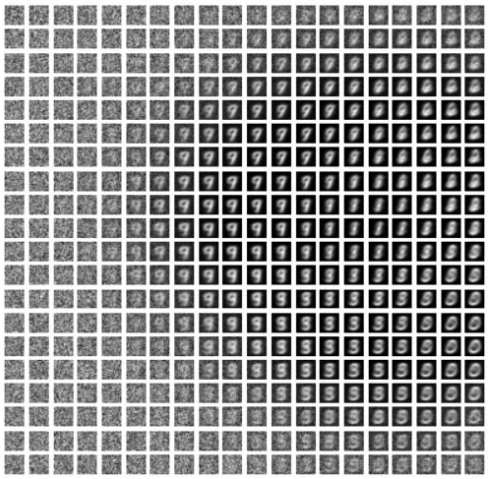


شکل ۹- epoch 10

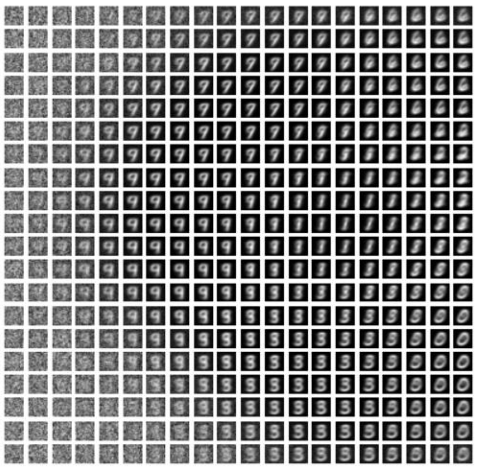
همان طور که مشاهده می‌کنید سرعت تغییر و converge شبکه به شکل قابل توجهی افزایش یافته‌است. بر خلاف حالت قبل در هر epoch تغییر اندکی نسبت به حالت قبل ایجاد می‌شد، اما اینجا تغییرات سریع‌تر انجام می‌شوند و در نهایت هم به نتیجه بهتری خواهیم رسید. شبکه‌ای در epoch آخر این حالت به آن رسیده‌ایم می‌توان تقریبا گفت converge شده‌است و به حالت ایده‌آل نزدیک کرده‌است. می‌توان گفت هر چه نرخ یادگیری بیشتر باشد، ما سریع‌تر به نتیجه دلخواه خود در kohonen خواهیم رسید.

۲-۱- تغییر شعاع همسایگی

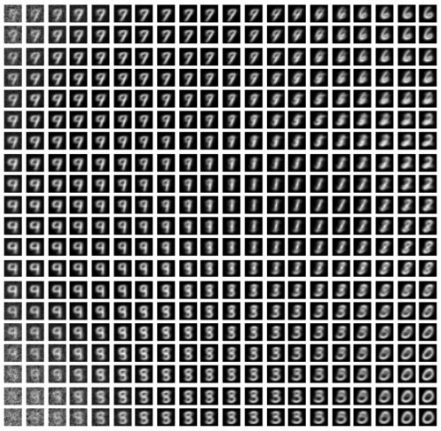
اگر مثال حالت قبل را با نرخ یادگیری 0.01 و این بار با شعاع ثابت انجام دهیم وزن‌های شبکه به صورت زیر خواهد بود:



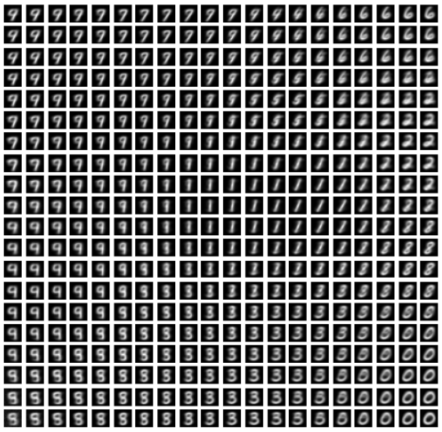
شکل ۱۳- epoch 1



شکل ۱۲- epoch 2



شکل ۱۵- epoch 5



شکل ۱۴- epoch 10

یکی از اثراتی که هنگامی که وزن‌های شبکه به طور بدی init شده‌اند و هم چنین شعاع ثابتی داریم ممکن است به وجود بیاید map distortion است. در این حالت وزن‌هایی که به شکل بدی init شده‌اند به همان شکل خود باقی می‌مانند یا حتی شکلی بدتر به خود می‌گیرند و هنگامی که شعاع نیز ثابت باشد همیشه نقاطی در یک فاصله ثابت تقویت می‌شوند که این شدت موضوع را بیشتر می‌کند.

در مثالی که آمده پخش شدن عدد ۹ در قسمت زیادی از شبکه می‌تواند نتیجه این اتفاق باشد که وزن‌های شبیه ۹ در قسمت زیادی بوده‌اند و هستند.

با استفاده از شعاع غیرثابت این اثر می‌تواند کاهش یابد که در دو مثال قسمت نرخ یادگیری تاثیر آن مشهود است.

۲- سوال دوم

در kohonen ما نیازی به داشتن برچسب‌ها نداریم زیرا این شبکه از استخراج خود ویژگی‌های داده‌ها و همچین ارتباط و شباهت میان آن‌ها سعی می‌کند که مشخص کند هر کدام در چه جایگاهی قرار می‌گیرند و استخراج این ویژگی‌های تاثیرگذار و هم چنین ارتباط ویژگی‌ها نقاط مختلف این شبکه را از نیاز به برچسب داده‌ها بی‌نیاز می‌کند.

برای اینکه بتوانیم از kohonen در classification نیز استفاده کنیم می‌توانیم به آن علاوه بر آن لایه وزن یک لایه خروجی یا label نیز اضافه کنیم. نورون برنده و ویژگی‌های آن می‌توانند به ما کمک کنند تا خروجی را نیز به دست آوریم. به عبارت دیگر می‌توانیم از وزن‌های نورون برنده به عنوان ویژگی‌های ورودی استفاده کنیم و با داشتن شبکه یا ساختاری مانند شبکه‌هایی که در قبل داشتیم مانند perceptron یا mlp برای تعیین label استفاده کنیم یا حتی راحت‌تر می‌توان گفت که خود مختصات آن نقطه می‌تواند برای ما label را با صورتی که گفته شد به دست آورد.

اگر یک داده‌ای که ندیده باشد را به آن بدهیم، ابتدا نورون برنده را برای آن ورودی به دست می‌آوریم سپس با مواردی که در آموزش گفته شد هر نورون به label اختصاص خواهد داشت با استفاده از label آن نورون که لایه خروجی آن را مشخص می‌کند می‌توانیم به label داده‌ای که تا به حال ندیدیم نیز پی ببریم.

۳- سوال سوم

خوشه‌بندی به عملیاتی گفته می‌شود که در آن نقطه‌های داده به چند گروه تقسیم می‌شوند که در آن نقطه‌هایی که به هم شباهت بیشتری دارند در یک گروه قرار می‌گیرند. Clustering یک روش unsupervised است. دو نوع خوشه‌بندی داریم که hard یا soft هستند. در hard یک نقطه تنها باید به یک cluster متعلق باشد در حالی که در soft یک نقطه می‌تواند متعلق به چند cluster باشد.

۳-۱- K-means

در این روش به تعداد دلخواه k خوشه در نظر می‌گیریم و به طور دلخواه k نقطه را به عنوان مرکز خوشه در نظر می‌گیریم.فاصله نقاطی که داریم و می‌خواهیم آن‌ها را خوشه‌بندی کنیم را تا مراکز cluster ها به دست می‌آوریم. هر نقطه فاصله کمتری تا یک مرکز cluster مشخص داشته باشد، به cluster آن مرکز تعلق خواهد داشت. پس از آنکه نقاط را به cluster ها نسبت دادیم، میانگین نقاط آن خوشه را خواهیم گرفت. این میانگین به عنوان مرکز جدید آن خوشه انتخاب خواهد شد. این کار تا زمانی انجام می‌شود که یا مراکز خوشه تغییرات کمی پیدا کنند یا اینکه پس از تعداد مشخص iteration این کار به پایان خواهد رسید.

DBSCAN -۳-۲

DBSCAN یا density-based spatial clustering of applications with noise یک روش خوشه‌بندی براساس density یا تراکم است که نقاط که به زیادی به هم نزدیک هستند را در یک گروه قرار می‌دهد. یک cluster در واقع یک منطقه high point density است که آن را نسبت به بخش‌های کم تراکم جدا می‌کند.

این الگوریتم دارای دو پارامتر است:

- minPts : حداقل نقاطی که با هم یک cluster را تشکیل می‌دهند و یک منطقه dense به وجود می‌آورند.
- eps : مقدار فاصله‌ای که برای مشخص کردن نقاط در همسایگی یک نقطه استفاده می‌شود.

پس از اینکه این الگوریتم به پایان برسد سه نوع نقطه وجود خواهد داشت:

- Core : این نقطه حداقل m نقطه با فاصله n از خودش دارد.
- Border : این نقطه حداقل یک Core در فاصله n از خود دارد.
- Noise : نقطه‌ای که نه Core است نه Border و کمتر از m نقطه با فاصله n ازخودش دارد.

مراحل این الگوریتم به صورت زیر است:

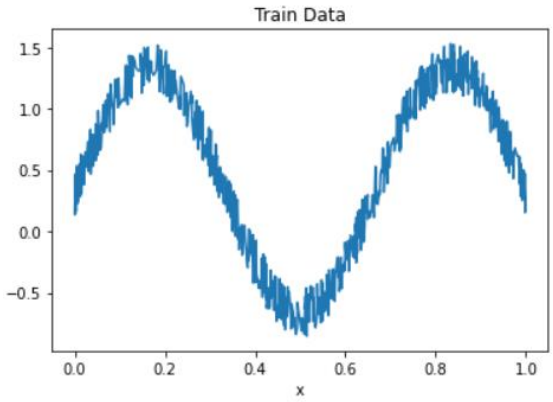
- یک نقطه به صورت تصادفی از مجموعه داده‌ها برمی‌داریم تا زمانی که تمامی نقاط دیده شوند.
- اگر حداقل minPts با فاصله eps از آن نقطه موجود بود، ما تمامی این نقاط را یک خوشه در نظر می‌گیریم.
- خوشه به تدریج با محاسبه همسایگی گسترش می‌یابد.

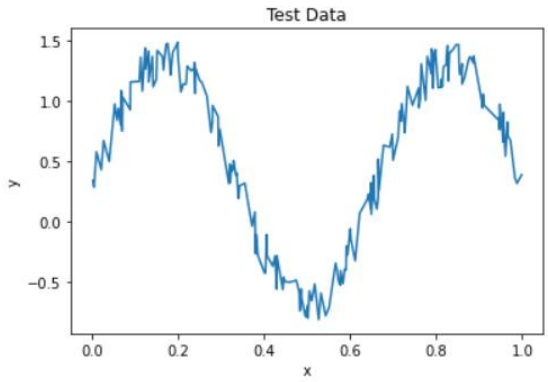
۴- سوال چهارم

برای ساختن مجموعه train و test ابتدا ۱۰۰۰ عدد رندوم در محدوده مشخص برای x و μ در نظر گرفتم و سپس مقادیر y بر اساس این مقادیر و خود تابع به دست آوردم. از این ۱۰۰۰ نمونه ۸۰۰ عدد را برای آموزش و ۲۰۰ عدد را برای test در نظر گرفتم:

```
NUM_SAMPLES = 1000
x = np.random.uniform(0., 1., NUM_SAMPLES)
x = np.sort(x, axis=0)
noise = np.random.uniform(-0.2, 0.2, NUM_SAMPLES)
y = 1/3 + np.sin(3 * np.pi * x)  + noise

idx = np.random.choice(np.arange(len(x)), 200, replace=False)
idx = np.sort(idx, axis=0)
x_test_plot = x[idx]
y_test_plot = y[idx]

x = np.delete(x, idx)
y = np.delete(y, idx)

x_train = np.reshape(x,(NUM_SAMPLES - 200,1))
y_train = np.reshape(x,(NUM_SAMPLES - 200,1))
x_test = np.reshape(x_test_plot,(200,1))
y_test = np.reshape(y_test_plot,(200,1))
```

نمودار داده‌های test و train به شکل‌های زیر خواهد بود:

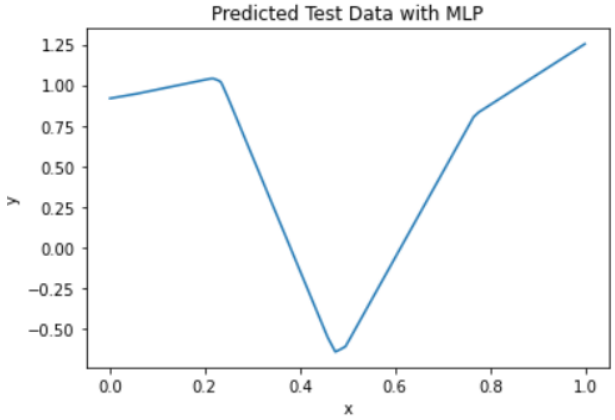




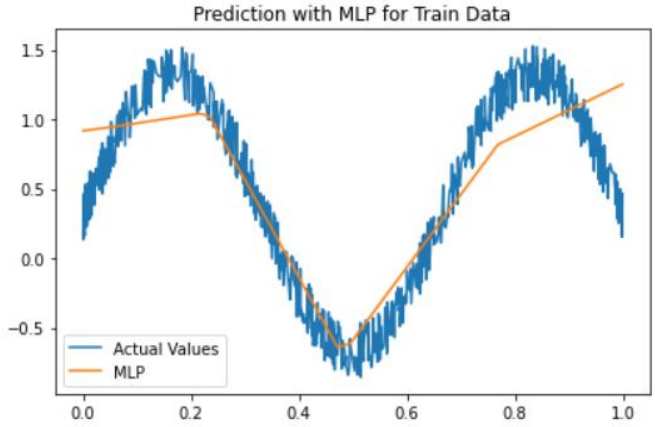
ابتدا سعی می‌کنیم با یک شبکه MLP این تابع را تقریب بزنیم. من از یک شبکه با مشخصات زیر برای آموزش استفاده کردم:

```
model = Sequential()
model.add(Dense(32, input_dim=1, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x_train, y_train, epochs=100, batch_size=128, shuffle=True)
```

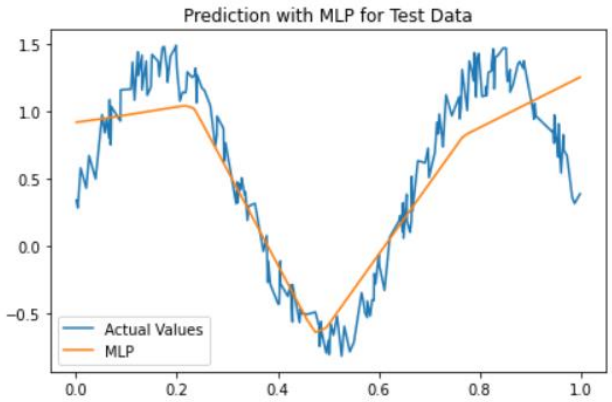
سپس از این شبکه train شده برای پیش‌بینی داده‌های تست استفاده می‌کنیم. نمودار زیر مقادیر پیش‌بینی‌شده توسط این شبکه برای داده‌های تست را نشان می‌دهد:



نمودار مقادیر واقعی و پیش‌بینی داده‌های train در mlp به صورت زیر است:



برای داده‌های test نیز نمودار به صورت زیر است:



حالا به سراغ RBF می‌رویم و کلاس متناظر با آن را پیاده‌سازی می‌کنیم. مواردی که باید برای آماده‌سازی اولیه این شبکه داشته باشیم k یعنی تعداد نورون‌های موجود در لایه میانی است. نرخ یادگیری و هم چنین الگوریتمی که برای init مراکز توابع شعاعی و width آن نیاز داریم موارد دیگر هستند. هم چنین وزن‌های نورون‌ها نیز در اینجا initialize می‌شوند. الگوریتمی که برای init مرکز استفاده می‌کنیم K-means است:

```
class RBF:

   def __init__(self, k, lr, center_algorithm='kmeans'):

     self.k = k
     self.lr = lr
     self.center_init = self.choose_algorithm(center_algorithm)
     self.w = np.random.rand(k, 1)

   def choose_algorithm(self, algorithm):
     if algorithm == 'kmeans':
       return self.K_means_Clustring
```

روش k-means را در سوال قبل توضیح دادیم ولی اینجا به طور دقیق‌تر به بررسی آن می‌پردازیم. در ابتدا از بین خود داده‌ها چند نقطه رندوم را به عنوان مراکز cluster انتخاب می‌کنیم. سپس هر بار نقاطی که به آن cluster تعلق دارند با میانگین خود مرکز جدید آن را مشخص می‌کنند. این کار یا تا زمان تمام شدن تعداد iteration ها یا converge شدن که وقتی اتفاق می‌افتد که مراکز cluster ها تغییرات نزدیک به صفر داشته باشند ادامه خواهد

یافت. پس از مشخص شدن مراکز مقدار width برای هر تابع شعاعی با مقدار انحراف معیار نقاط آن cluster تعیین خواهد شد:

```
def K_means_Clustring(self, x, max_iteration=300):
  centers = np.random.choice(np.squeeze(x), size=self.k)
  prevCenters = centers.copy()
  converged = False
  current_iteration = 0
  clusters_points = [None] * self.k

  while (not converged) and (current_iteration < max_iteration):
       distances = np.abs(x - centers)
       closestCluster = np.argmin(distances, axis=1)
       for i in range(self.k):
           pointsForCluster = x[closestCluster == i]
           if len(pointsForCluster) > 0:
               centers[i] = np.mean(pointsForCluster, axis=0)
               clusters_points[i] = pointsForCluster

       convergance_number = np.float32(1e-4)
       converged = np.sum(np.abs(prevCenters - centers)) < convergance_number
       prevCenters = centers.copy()
       current_iteration += 1
```

یکی دیگر توابع موجود در این کلاس خود تابع rbf است که با داشتن نقطه موردنظر و مرکز آن تابع شعاعی و هم چنین width آن به محاسبه مقدار آن می‌پردازد:

```
def rbf(self, x, c, s):
 return np.exp((-1 * (x-c)**2) / s**2)
```

در نهایت تابع fit را داریم که در آن به آموزش شبکه می‌پردازیم:

```
def fit(self, X, y, epochs):
  self.centers, self.widths = self.center_init(X)
  for epoch in range(epochs):
     print(f'epoch {epoch+1}/{epochs}:')
     for i in range(X.shape[0]):
         a = np.array([self.rbf(X[i], c, s) for c, s, in zip(self.centers, self.widths)])
         approximation = np.sum(a * self.w)
         loss = (y[i] - approximation).flatten() ** 2
         if (i + 1) % 100 == 0:
           print(f'{i+1}/{X.shape[0]}    Loss: {loss}')
         error = np.sum(-(y[i] - approximation))
         self.w = self.w - self.lr * a * error
```

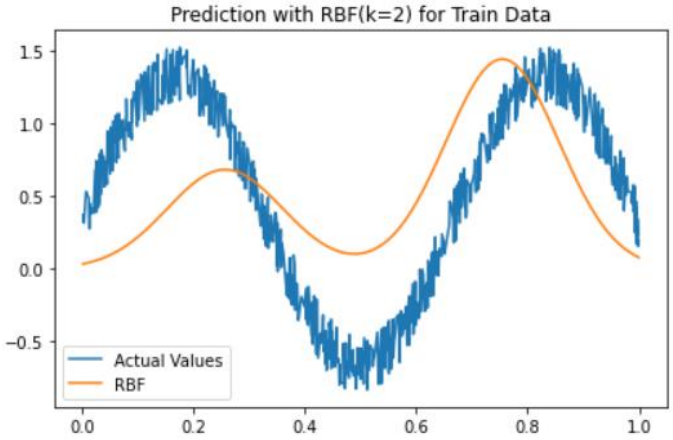
لازم است برای هر نقطه از داده توابع شعاعی متناظر با مراکز و width های مختلف محاسبه شود و ضرب این موارد در وزن‌های شبکه خروجی را به ما خواهد داد. حالا لازم است با مقایسه با مقدار y اصلی ببینیم که مقدار اصلی ببینیم که مقدار error ما به چه میزان است. با داشتن میزان خطا، نرخ یادیگری و هم چنین مقادیر توابع شعاعی می‌توانیم تغییرات وزن را به دست آوریم.

ابتدا سعی می‌کنیم RBF را با k=2 روی داده‌های train این شبکه را آموزش دهیم:

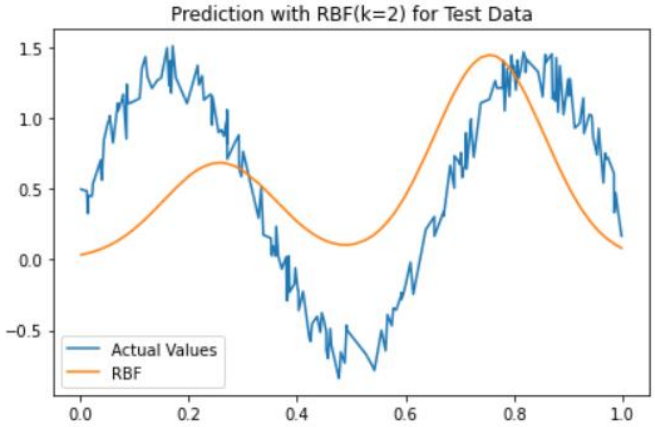
```
RBF_Network = RBF(2, 0.01, 'kmeans')
RBF_Network.fit(x_train, y_train, 100)
```

```
500/800    Loss: [0.25205079]
600/800    Loss: [0.00267923]
700/800    Loss: [0.16573645]
800/800    Loss: [0.01270539]
epoch 95/100:
```

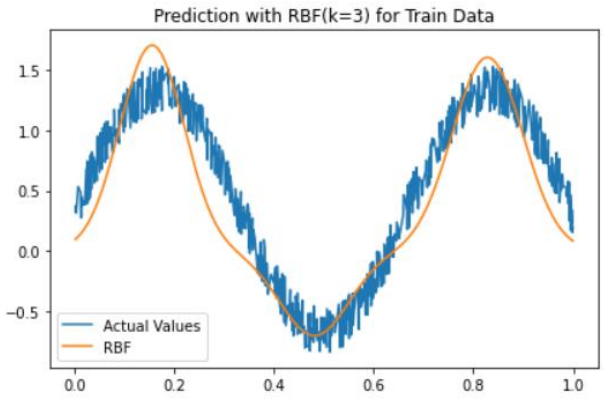
نمودار مقادیر پیش‌بینی‌شده و اصلی برای داده‌های train به صورت زیر است:



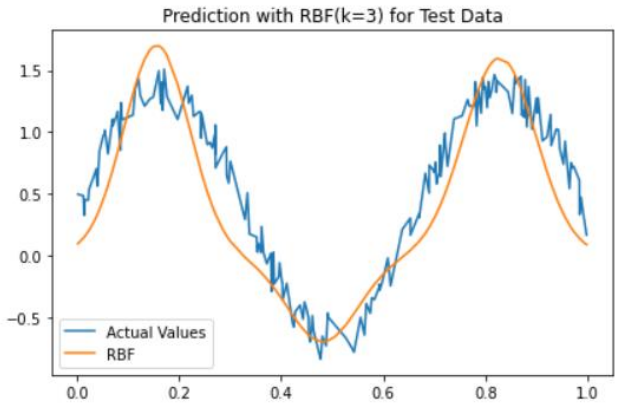
و نمودار برای test هم به شکل زیر است:



اگر این بار k=3 باشد برای train به شکل زیر است:

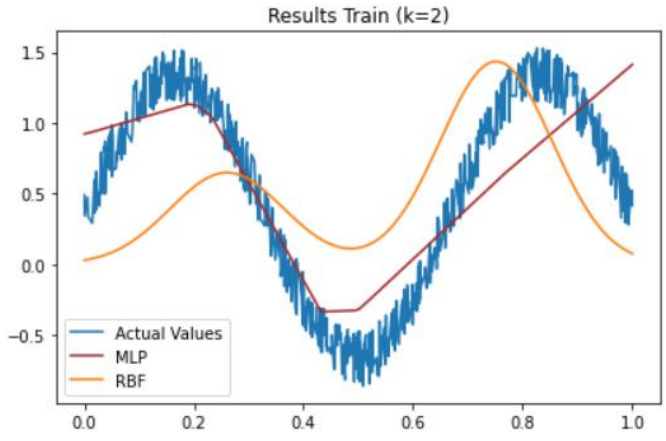


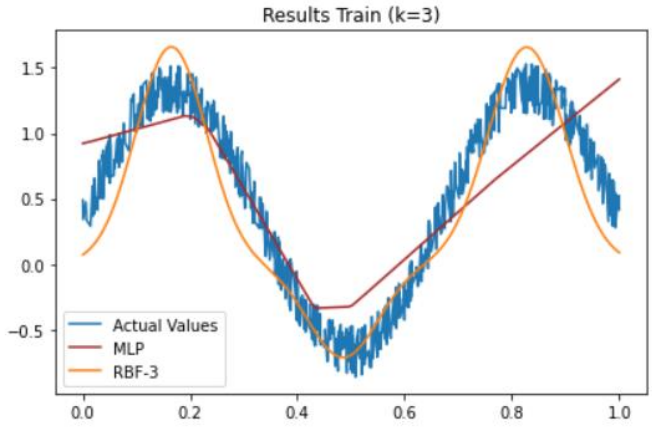
و برای test نیز به شکل زیر است:



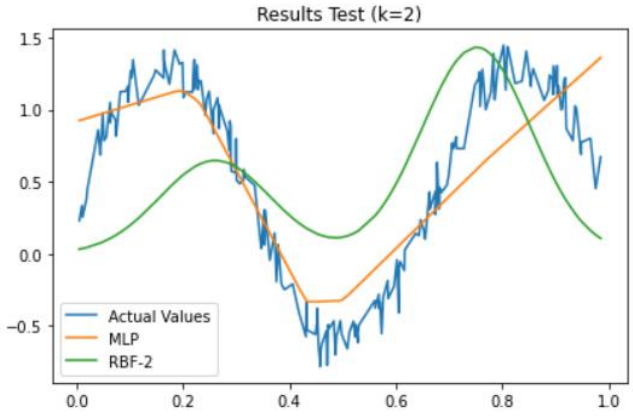
حالا به مجموع نتایج کلی می‌پردازیم:

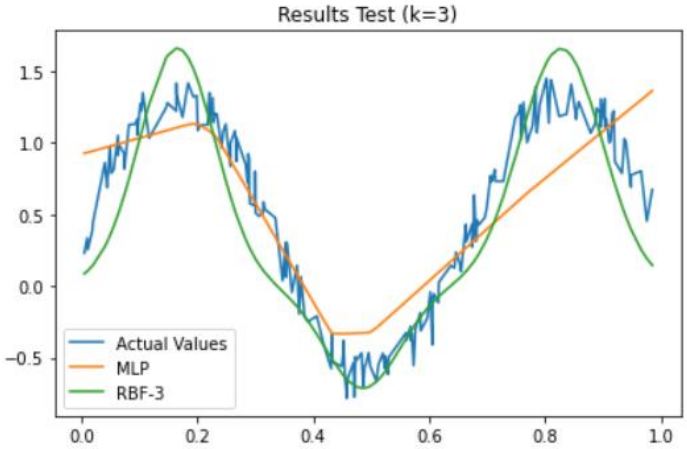
نتایج کلی داده‌های train به شکل زیر هستند:





و نتایج داده‌ها test نیز به شکل زیر است:





MLP معمولا نمی‌تواند به خوبی نویز را در داده‌ها مدیریت کند و راه حلی ارائه بدهد که برای داده‌ها با نویزهای مختلف پاسخگو باشد. در نمودارها هم این مورد مشهود است که MLP در آن سعی می‌کند خطی که در میانگین داده‌ها ارائه دهد که میان نویزها باشد.

اما RBF در این کار بهتر عمل می‌کند و سعی می‌کند نتیجه‌ای ارائه دهد که برای داده‌های مختلف دارای نویز بیشتر هم کارساز باشد. تنظیم k در RBF بسیار مهم است و می‌تواند باعث یک نتیجه خوب یا بد شود. RBF برای کاربردهای function approximation کاربرد بیشتری نسبت به RBF دارد در حالی که MLP بیشتر می‌تواند در classification به ما کمک کند.

با این حال اگر k به درستی انتخاب نشود ممکن است نتایج MLP به مراتب بهتر از RBF باشد چون k نامناسب می‌تواند باعث ایجاد تخمینی نادرست از تابع شود.

می‌توان شبکه‌ای داشت که بعضی از لایه‌های آن rbf و بعضی mlp باشد به صورتی که برای مثال خروجی توابع rbf در وزنشان ورودی‌های دیگری برای لایه‌های mlp می‌شوند. اما در مورد خروجی از mlp و ورودی به rbf شاید قضیه کمی پیچیده‌تر به نظر بیاید چون مرکز و width برای ما مشخص نیست. شاید یک راه این باشد که ابتدا یک مرکز رندوم در نظر بگیریم و با استفاده داده‌هایی که هر بار به آن وارد می‌شوند و جمع آوری آن‌ها پس از یک iteration کامل روی داده، سراغ k-means برویم و سعی کنیم با آن‌ها مراکز را تعیین کنیم. به این ترتیب می‌توانیم شبکه‌ای متشکل از mlp و rbf داشته باشیم.