۱-کدهای مربوط به این سوال در فایل Q1.ipynb نوشته شده اند.
ابتدا کتابخانه و ابزار لازم را فراخوانی کردیم.

1) import modules and mnist dataset.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from tqdm.notebook import tqdm
import math
import tensorflow as tf
import tensorflow.keras as keras
from keras.datasets import mnist
from keras import models
from keras import layers
from tensorflow.keras.utils import to_categorical
import random as rnd
```

سپس با توجه با تابع داده شده ی زیر،

$$y = -1 + \left(\frac{2}{3}\right) * \sin(2x * pi) + L$$
$$0 < x < 2$$
$$-0.8 < L < 0.8$$

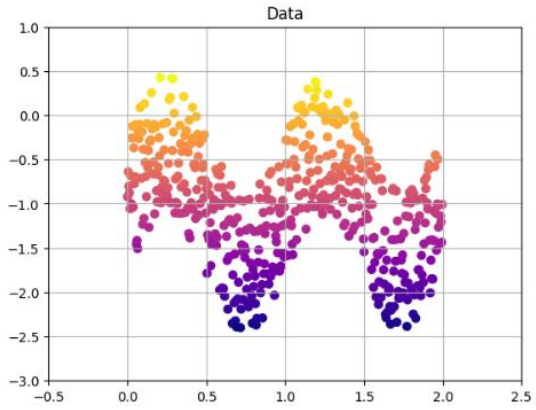
مجموعه ی x و y را با کمک کتابخانه ی Numpy، ساختیم.
۸۰ درصد داده های ساخته شده را برای آموزش و ۲۰ درصد را برای تست در نظر گرفتیم.
بازه ی داده شده برای نویز تابع بین -۰,۸ و ۰,۸ بود که در نظر گرفتم.

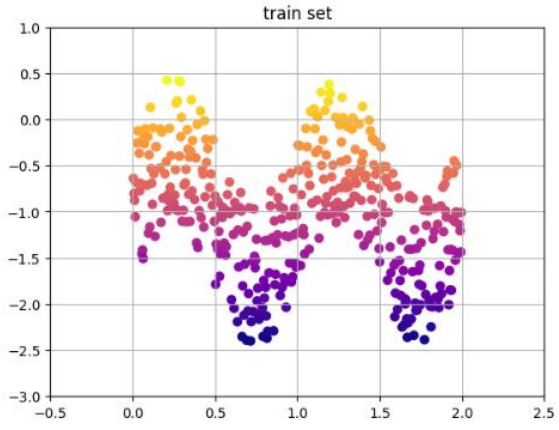
2)make data by given function and normalize

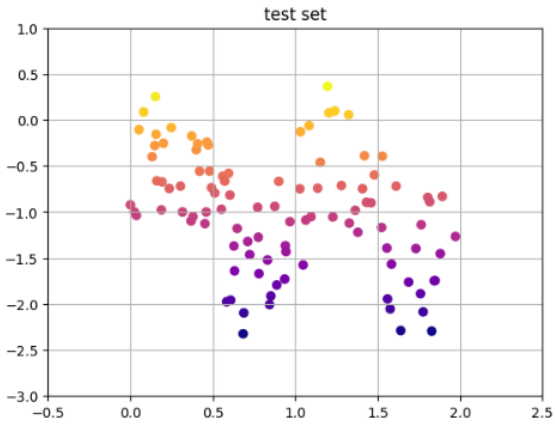
```
np.random.seed(0)


def function(x):
    u = 1.6 * np.random.random_sample((500)) - 0.8
    y = (-1) + (2/3) * (np.sin(2 * x * np.pi)) + u
    return y
n1 = 100
n_train1 = 80
n_test1 = 20
x1 = np.arange(0, 2, 0.004)
y1 = function(x1)

indexes = np.random.choice(np.arange(500), 400, replace=False)
indexes = np.sort(indexes, axis = 0)

x_train1 = x1[indexes]
y_train1 = y1[indexes]

x_test1 = np.delete(x1.copy(),indexes)
y_test1 = np.delete(y1.copy(),indexes)
```

3)show the data

```
plt.scatter(x1, y1, c=y1, cmap='plasma')
plt.grid()
plt.xlim(-0.5, 2.5)
plt.ylim(-3, 1)
plt.title('Data')
plt.show()

plt.scatter(x_train1, y_train1, c=y_train1, cmap='plasma')
plt.grid()
plt.xlim(-0.5, 2.5)
plt.ylim(-3, 1)
plt.title('train set')
plt.show()

plt.scatter(x_test1, y_test1, c=y_test1, cmap='plasma')
plt.grid()
plt.xlim(-0.5, 2.5)
plt.ylim(-3, 1)
plt.title('test set')
plt.show()
```



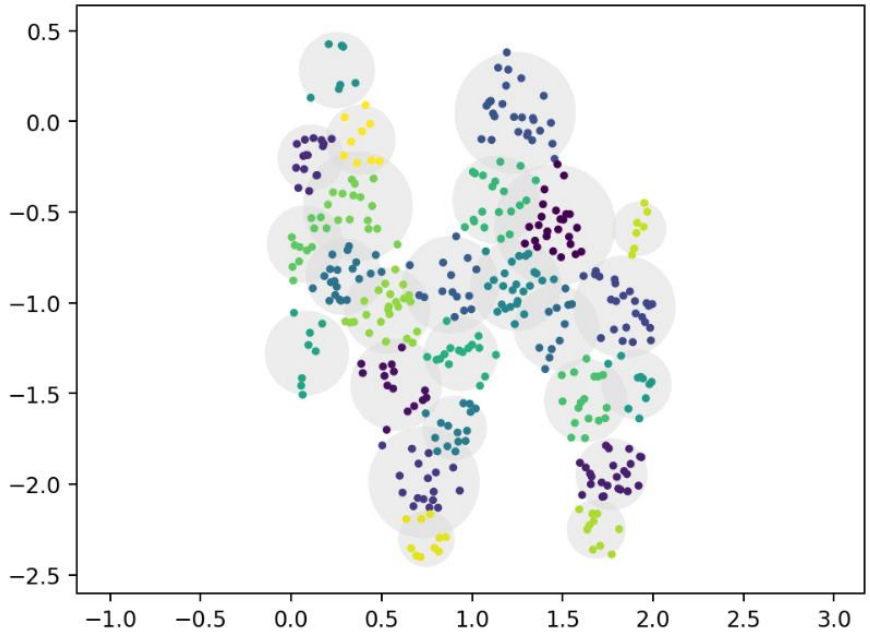




در ادامه ی سوال ابتدا مدل K-means را از کتابخانه ی sklearn صدا زدم و مدل ساختم و بعد با امتحان کردن تعداد کلاس های مختلف سعی کردم تعداد کلاس مناسب برای مرکز ها در این روش را پیدا کنم.

4)K-Means and GMM

```
from numpy import random
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

X = np.array(list(zip(x_train1, y_train1)))

#keams
from scipy.spatial.distance import cdist
def plot_kmeans(kmeans, X, ax=None):
    labels = kmeans.fit_predict(X)
# plot the input data
    plt.figure(dpi=175)
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=7, cmap='viridis', zorder=2)
# plot the representation of the k-means model
    centers = kmeans.cluster_centers_
    radii = [
        cdist(X[labels == i], [center]).max()
        for i, center in enumerate(centers)
    ]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(
            c, r, fc='#DDDDDD', lw=3, alpha=0.5, zorder=1
        ))


kmeans = KMeans(n_clusters=25, random_state=0).fit(X)
labels_knn = kmeans.predict(X)
plot_kmeans(kmeans, X)
```

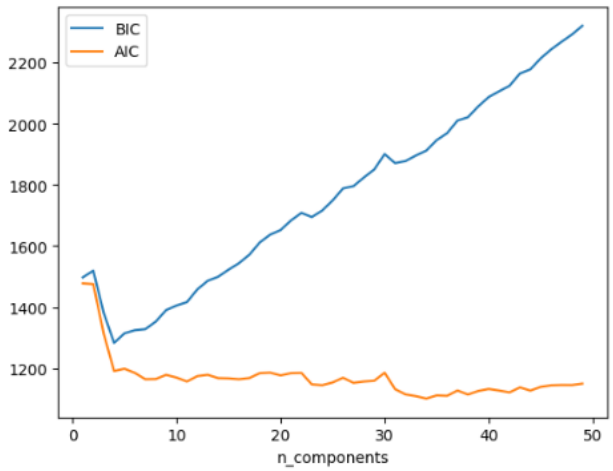


همانطور که در شکل مشخص شده است ۲۵ مرکز دایره طوری انتخاب شده اند که با شعاع های مناسب بیشترین همپوشانی در دیتا را داشته باشند.

سپس برای مدل GMM همینکار را کردیم با این تفاوت که تعداد خوشه های متفاوت را امتحان کردیم و بهترین را روی شکل نشان دادیم.

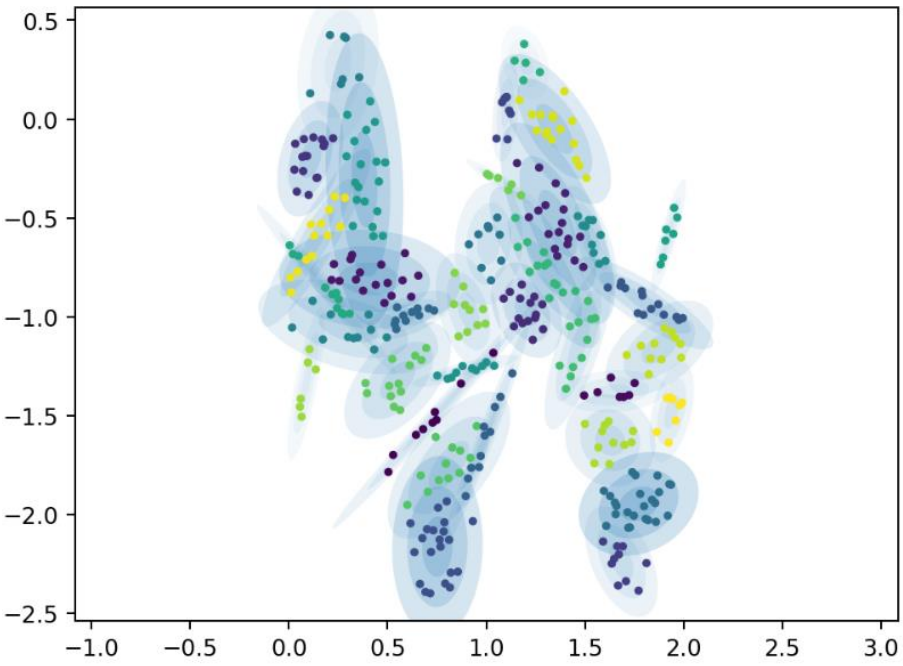
```
from sklearn.mixture import GaussianMixture

#Gmm
from matplotlib.patches import Ellipse
def plot_gmm(gmm, X, label=True, ax=None):
    def draw_ellipse(position, covariance, ax=None, **kwargs):
        ax = ax or plt.gca()
        if covariance.shape == (2, 2):
            U, s, Vt = np.linalg.svd(covariance)
            angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
            width, height = 2 * np.sqrt(s)
        else:
            angle = 0
            width, height = 2 * np.sqrt(covariance)
        for nsig in range(1, 4):
            ax.add_patch(Ellipse(
                position, nsig * width, nsig * height,
                angle, **kwargs
            ))

    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=7, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=7, zorder=2)

    ax.axis('equal')
    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)

n_components = np.arange(1, 50)
models = [GaussianMixture(n, covariance_type='full', random_state=0).fit(X)
          for n in n_components]

plt.plot(n_components, [m.bic(X) for m in models], label='BIC')
plt.plot(n_components, [m.aic(X) for m in models], label='AIC')
plt.legend(loc='best')
plt.xlabel('n_components');
```

طبق نمودار بدست آمده ی زیر بهینه ترین حالت وقتی است که به دنبال ۳۰ تا ۳۵ مرکز باشیم. البته از حدود ۴ مرکز به بعد تفاوت ندانی نمیکند.



سپس نتیجه ی اجرای مدل روی دیتای آموزشی را نمایش میدهیم.

```
gmm = GaussianMixture(n_components=34, covariance_type='full', random_state=42).fit(X)
labels_gmm = gmm.predict(X)
plt.figure(dpi=175)
plot_gmm(gmm, X)
```



همانطور که مشخص شده است سعی شده بهترین مرکز ها برای پوشش دادن همه ی نقاط به صورت گوسی انتخاب شوند.
اما برای مقایسه ی بهتر بین روش های رندوم، k-mean و GMM بهتر است آن ها را با تعداد مرکز برابر و شرایط مشابه روی داده های آموزشیُ آموزش دهیم و مدل های ساخته شده را روی داده های تست امتحان کنیم و همین عملیات را برای MLP هم تکرار میکنیم.
برای مدل های RBF من ۲۰ مرکز را انتخاب میکنم. سپس توابع مورد نیاز train و predict را هم پیاده سازی کردم. همچنین نرخ یادگیری را ۰٫۰۱ و تعداد دوره های تکرار را ۵۰۰ درنظر گرفتم.

5)Complete RBF Model

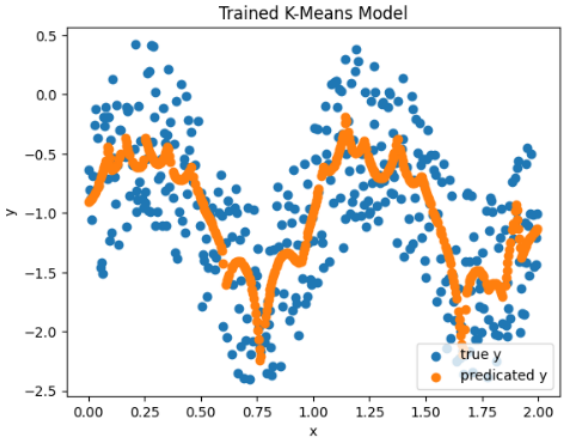
```
import scipy.stats
train_data_number = 400
center_number = 30
learning_rate = 0.01

class RBF:
    def __init__(self, data_number, center_number, method, lr):
        self.data_number = data_number
        self.center_number = center_number
        self.lr = lr

        self.method = method
        self.b = np.random.random((1,1))
        self.w = np.random.random((self.center_number,1))

    def train(self, data_inp, data_out, epochs, flag=False):
        if (flag == False):
            if (self.method == "random"):
                self.centers = self.Randomly(data_inp)
            elif (self.method == "kmeans"):
                self.centers = self.K_means(data_inp, data_out)
            elif (self.method == "gmm"):
                self.centers = self.GMM(data_inp, data_out)

            max_dist = np.max([np.abs(c1 - c2) for c1 in self.centers for c2 in self.centers])
            l1 = max_dist / np.sqrt(2*self.center_number)
            self.rs = np.repeat(l1, self.center_number)
            self.rs = np.expand_dims(self.rs, axis=1)

        for e in range(epochs):
            for i in range(self.data_number):
                self.produce_output(data_inp[i], data_out[i])

    def Randomly(self, data):
        indexes = random.permutation(len(data))
        indexes = indexes[:self.center_number]
        centers = [data[i] for i in indexes]
        return np.array(centers)

    def K_means(self, data_inp, data_out):
        data = list(zip(data_inp, data_out))
        centers = KMeans(n_clusters=self.center_number).fit(data)
        centers = np.array([centers.cluster_centers_[i][0] for i in range(len(centers.cluster_centers_))])
        return centers

    def GMM(self, data_inp, data_out):
        data = list(zip(data_inp, data_out))
        gmm = GaussianMixture(n_components=self.center_number)
        gmm.fit(data)
        centers = []
        for i in range(gmm.n_components):
            covar = gmm.covariances_[i]
            mean = gmm.means_[i]
            status = scipy.stats.multivariate_normal(cov=covar, mean=mean)
            density = status.logpdf(data)
            centers.append(data_inp[np.argmax(density)])
        return np.array(centers)

    def predict(self, x):
        y = []
        for i in range(len(x)):
            y.append(self.produce_output(x[i], y_train=None))
        return np.array(y)

    def produce_output(self, data, y_train):
        self.centers = self.centers.reshape(self.center_number, 1)
        distance = -np.abs(data-self.centers)
        rbf_res = np.exp(distance/np.power(self.rs,2))
        output = np.dot(self.w.T, rbf_res) + self.b
        if (y_train is not None):
            error = output.reshape(1,1) - y_train.reshape(1,1)
            self.w = self.w - self.lr * rbf_res * error
            self.b = self.b - self.lr * error
        return output
```

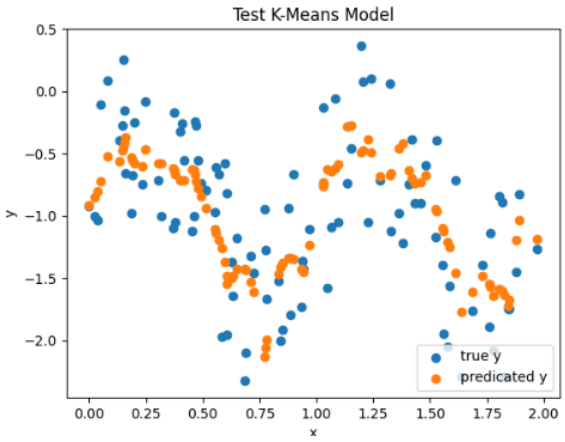
ابتدا مدل K-Means را ساختم و آن را روی دیتای آموزشی ترین کردم سپس نتایج دیتای تست را بدست آورده و با نتایج درست مقایسه کردم.

6)Train K-Means

```
K_Means_Model = RBF(train_data_number, center_number, "kmeans", learning_rate)
K_Means_Model.train(x_train1, y_train1, 500)

K_Means_Model_Predict_Labels = K_Means_Model.predict(x_train1).reshape((-1,1))

plt.scatter(x_train1, y_train1)
plt.scatter(x_train1, K_Means_Model_Predict_Labels)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Trained K-Means Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

K_Means_Model_Predict = K_Means_Model.predict(x_test1).reshape((-1,1))

plt.scatter(x_test1, y_test1)
plt.scatter(x_test1, K_Means_Model_Predict)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Test K-Means Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

در نمودارهای زیر نقاط آبی رنگ نقاط در مجموعه های آموزشی و تست هستند و نقاط نارنجی رنگ، نقاطی هستند که xآنها همان مقدار x نقاط آبی رنگ است اما y آنها را مدل ما پیشبینی کرده است. مثلا اگر دیتای جدیدی هم تولید کنیم و آن را در مجموعه ی تست قرار دهیم مقدار y آن را طوری محاسبه میکند که آن نقطه روی خط نارنجی رنگ در نمودار بالاتر میفتد. علت اینکه در نمودار بالاتر تر نقاط نارنجی به هم متصل دیده میشوند تعداد زیاد نقاط است.



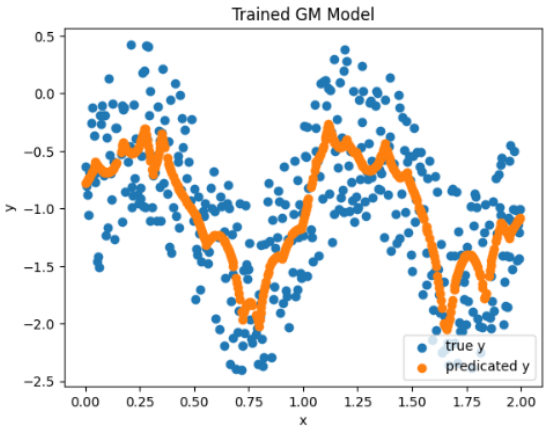


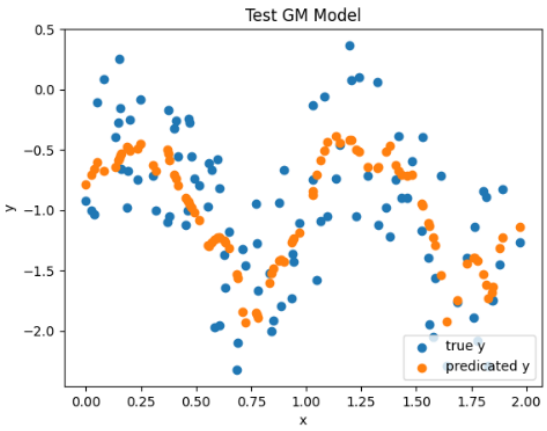
همین مراحل را برای GMM هم تکرار کردیم.

7)Train GM Model

```
GM_Model = RBF(train_data_number, center_number, "gmm", learning_rate)
GM_Model.train(x_train1, y_train1, 500)


GM_Model_Predict_Labels = GM_Model.predict(x_train1).reshape((-1,1))

plt.scatter(x_train1, y_train1)
plt.scatter(x_train1, GM_Model_Predict_Labels)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Trained GM Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

GM_Model_Predict = GM_Model.predict(x_test1).reshape((-1,1))

plt.scatter(x_test1, y_test1)
plt.scatter(x_test1, GM_Model_Predict)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Test GM Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

نتایج آموزش و تست این مدل در نمودارهای زیر قابل مشاهده هستند.



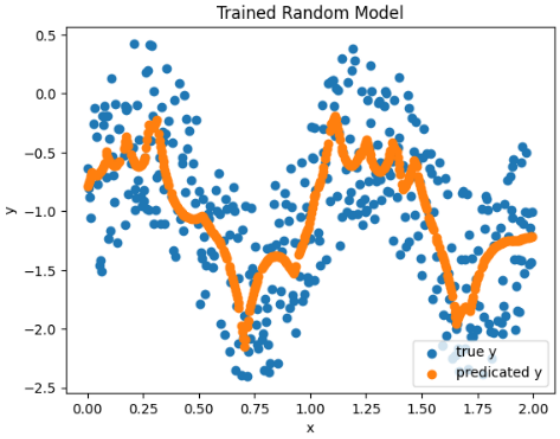


تفسیر این نمودار هم مانند قبلی است. یعنی نقاط نارنجی پیش بینی شده ها هستند.

دقیقا همین مراحل را برای مدل رندوم انجام دادیم و نتایج را ثبت کردیم.

7)Train Random Model

```
Random_Model = RBF(train_data_number, center_number, "random", learning_rate)
Random_Model.train(x_train1, y_train1, 500)


Random_Model_Predict_Labels = Random_Model.predict(x_train1).reshape((-1,1))

plt.scatter(x_train1, y_train1)
plt.scatter(x_train1, Random_Model_Predict_Labels)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Trained Random Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

Random_Model_Predict = Random_Model.predict(x_test1).reshape((-1,1))

plt.scatter(x_test1, y_test1)
plt.scatter(x_test1, Random_Model_Predict)
plt.legend(["true y", "predicated y"], loc ="lower right")
plt.title("Test Random Model")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



تفاوت رندوم با الگوریتم های دیگر این است که مرکز تابع شعاعی را به صورت ارندوم از بین خود نقاط آموزشی انتخاب کرده است.

حالا به سراغ پیاده سازی مدل MLP میرویم.

من با توجه به مثلثاتی بودن تابع و پیچیدگی آن، ۳ لابه برای این مدل درنظر گرفتم که به ترتیب دارای نورون های ۲۰۰و ۱۰۰و ۵۰ هستند و همگی fully connected هستند. همچنین برای لایه ی اکتیویشن هر کدام تابع ساده ی relu را فرض کردم و برای بهینه سازی از روش SGDاستفاده کردم. همچنین تابع ضرر را هم MSE فرض کردم.

8)MLP

```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.models import Sequential

l0 = Input(shape=(1))

l1 = Dense(200)(l0)
l1 = Activation("relu")(l1)

l2 = Dense(100)(l1)
l2 = Activation("relu")(l2)

l3 = Dense(50)(l2)
l3 = Activation("relu")(l3)

l4 = Dense(1)(l3)

MLP = Model(inputs=l0, outputs=l4)
MLP.compile(loss='mean_squared_error', optimizer='SGD')
MLP.summary()
```

خلاصه اطلاعات مدل ساخته شده به شرح زیر است:

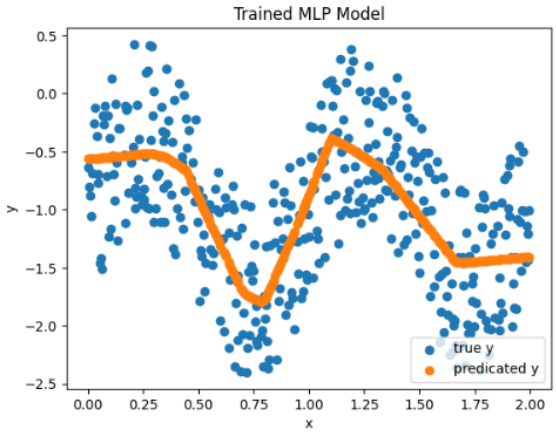
```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 1)]               0

 dense_12 (Dense)            (None, 200)               400

 activation_9 (Activation)   (None, 200)               0

 dense_13 (Dense)            (None, 100)               20100

 activation_10 (Activation)  (None, 100)               0

 dense_14 (Dense)            (None, 50)                5050

 activation_11 (Activation)  (None, 50)                0

 dense_15 (Dense)            (None, 1)                 51

=================================================================
Total params: 25,601
Trainable params: 25,601
Non-trainable params: 0
_________________________________________________________________
```

مدل MLP را هم با همان ۴۰۰ دیتای آموزشی fit میکنیم و سپس نتایج دیتای تست را ثبت می نماییم.

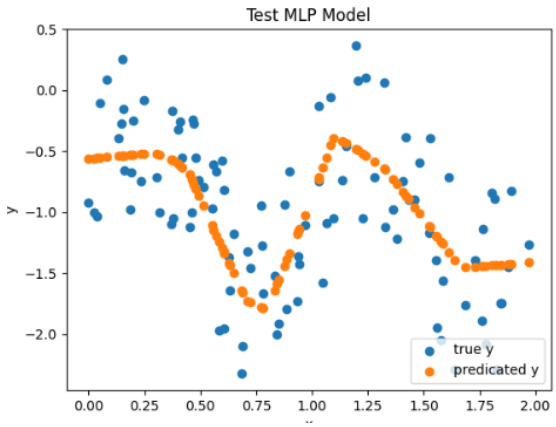
```
data = np.array(list(zip(x_test1, y_test1)))
MLP.fit(x_train1, y_train1, validation_data=(x_test1, y_test1), epochs=800, batch_size=64)
Epoch 792/800
7/7 [==============================] - 0s 10ms/step - loss: 0.2159 - val_loss: 0.2198
Epoch 793/800
7/7 [==============================] - 0s 9ms/step - loss: 0.2179 - val_loss: 0.2163
Epoch 794/800
7/7 [==============================] - 0s 10ms/step - loss: 0.2147 - val_loss: 0.2185
Epoch 795/800
7/7 [==============================] - 0s 9ms/step - loss: 0.2139 - val_loss: 0.2148
Epoch 796/800
7/7 [==============================] - 0s 11ms/step - loss: 0.2189 - val_loss: 0.2191
Epoch 797/800
7/7 [==============================] - 0s 11ms/step - loss: 0.2139 - val_loss: 0.2137
Epoch 798/800
7/7 [==============================] - 0s 11ms/step - loss: 0.2140 - val_loss: 0.2515
Epoch 799/800
7/7 [==============================] - 0s 11ms/step - loss: 0.2247 - val_loss: 0.2356
Epoch 800/800
7/7 [==============================] - 0s 11ms/step - loss: 0.2173 - val_loss: 0.2123

<keras.callbacks.History at 0x1d03b3dae20>
```

نمودار پیدا شده از نقاط پیشبینی شده توسط MLP هم به صورت زیر شد:

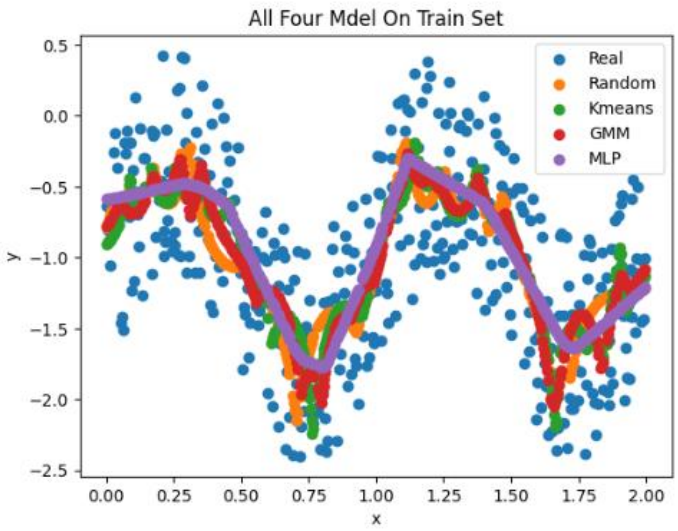
```
13/13 [==============================] - 0s 2ms/step
```

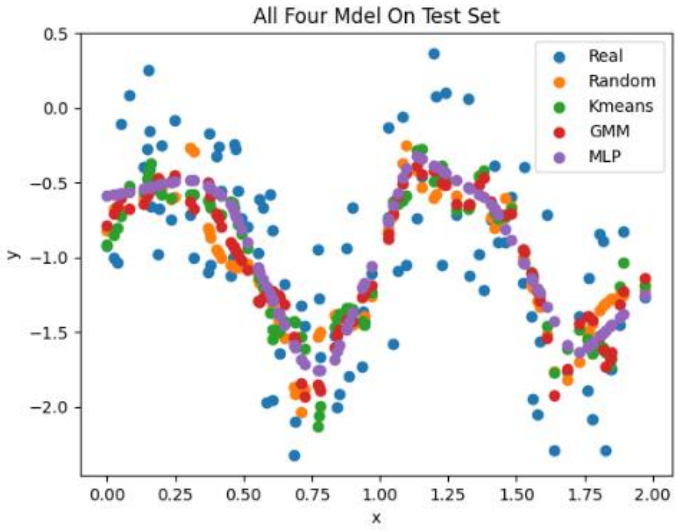



```
4/4 [==============================] - 0s 2ms/step
```



حالا باید جواب ها را با هم مقایسه کنیم:





مقایسه:

بهترین مدل روی این دیتا که دارای نویز است مدل RBF است که از توابع سعاعی استفاده میکند و محدودی زیادی از نقاط را میتواند با تقریب خیلی خوبی پیدا کند و همچنین قابلیت تعمیم بهتری هم دارد یعنی نقاطی را که آموزش ندیده به خوبی پیدا میکند.

شبکه ی MLP خخطی را انتخاب میکند که تقریبا وسط نقاط است یعنی دو طرف آن نقاط به یک تعداد هستند. در مدل های RBF به تراکم نقاط هم توجه شده و برای همین در بعضی جاها منحنی چند بار بالا و پایین شده است.

در RBF میتوانیم منحنی های پیچیده را با چند مرکز پیدا کنیم مثلا منحنی حلزونی شکل با دو تابع شعاعی قابل حل است اما همین مسئله نیاز به MLP خیلی عمیقی دارد. RBF با پیدا کردن نواحی و مینیمم فاصله ی نقاط دامنه تابع را پیدا میکند اما MLP مستقیما با چند مدل پرسپترون ناحیه ی محدب را پیدا میکند و اگر جواب به طور مستقیم ناحیه ی محدب نباشد باید فضای مسئله را تغییر داد تا بتواند حل کند. برای همین در مسائل اینگونه که پیچیده هستند MLP بیشتر طول میکشد و از نظر سرعت کند تر است.
از طرفی در بین ه مدل RBF گفته شده انتخاب رندوم مرکز های توابع شعاعی خیلی مناسب نیست. از بین خود نقاط مجموعه انتخاب میشود و به سختی نقاط نویزی و دور تر را پیدا میکند و قابلیت تعمیم کمتری دارد.

بعد از آن K_Means است که دقت خیلی خوبی دارد اما باید در این حال های نویزی متغیر K را با تعداد مناسبی انتخاب کنیم تا نقاط به خوبی پیدا شوند.
در مدل GMM تابع گوسی پراکندگی نقاط را در نظر میگیرد و خطی که پیدا میکند به جایی که تراکم نقاط بیشتر است نزدیک تر است در نتیجه نویز نقاط را هم درنظر میگیرد.
از کاربرد الگوریتم K-Means می توان به موارد زیر اشاره کرد:

- سنجش عملکرد تحصیلی) بر اساس نمرات، دانش آموزان در نمرات A ، Bیا C طبقه بندی می‌شوند.(
- سیستم‌های تشخیصی (سیستم های تشخیصی حرفه ای پزشکی در ایجاد سیستم های پشتیبانی دقیق تصمیم پزشکی، به ویژه در درمان بیماری های کبدی، شناسایی بیماری قلبی و شناسایی تومرها و بررسی تاثیر داروها)
- موتورهای جستجو
- شبکه های حسگر بی سیم
- تقسیم بازار (تقسیم بندی بازار، تقسیم بندی مشتریان به منظور تعیین مشتری هدف و تبلیغات هدفمند وتاثیرگذار)
- تقسیم تصویر و فشرده سازی تصویر و پردازش تصویر

مزایا الگوریتمK-Means

روش خوشه بندی k-means ساده ترین روش برای خوشه بندی داده ها است. از مزایا الگوریتم K-Means می‌توان به موارد زیر اشاره کرد:

- سرعت بسیار بالای این روش در اجرا.
- استفاده از این الگوریتم بسیار ساده (با استفاده از کتابخانه ها و پکیج های اماده مربوط به این الگوریتم) است.
- این الگوریتم برای داده هایی با حجم زیاد قابل استفاده است.
- این الگوریتم همگرایی را تضمین می‌کند.
- این الگوریتم بهترین موقعیت ها برای مراکز را در نظر می‌گیرد.
- این روش به راحتی با نمونه های جدید سازگار می‌شود.
- خوشه هایی با شکل و اندازه های مختلف مانند خوشه های بیضوی تعمیم می‌یابد.

معایب الگوریتمK-Means

از معایب الگوریتم K-Means می‌توان به موارد زیر اشاره کرد:

- نیاز داشتن به تعیین تعدادخوشه ها.
- این الگوریتم دارای دقت کم در داده ها با شکل غیر محدب است.

مزایای پرسپترون چند لایه

- مزایای پرسپترون چند لایه عبارتند از:
- قابلیت یادگیری مدل های غیر خطی
- قابلیت یادگیری مدل ها در زمان واقعی (آموزش آنلاین).

معایب پرسپترون چند لایه

- معایب پرسپترون چند لایه عبارتند از:
- پرسپترون چند لایه با لایه های پنهان دارای تابع زیان غیر محدب (non-convex loss function)است که در آن بیش از یک کمینه محلی وجود دارد. بنابراین مقداردهی های مختلف وزن تصادفی می تواند منجر به دقت اعتبارسنجی شود.
- MLPنیاز به تنظیم تعدادی از پارامترهای فوق العاده مانند تعداد سلول های عصبی پنهان، لایه ها و تکرارها دارد.

- MLPبه مقیاس بندی ویژگی ها حساس است.

درRBF ، کاربر اساسا مجبور است به ازای هر پیچیدگی که در «داده‌ها (Data) «وجود دارد، یک «نورن (Neuron) «اضافه کند و این نورن‌ها یک «موازنه (Trade-Off) «دارند و فرد اگر بخواهد آن را خیلی دقیق کند مجبور می‌شود تعداد نورن‌ها و پیچدگی مدل را افزایش دهد و اگر بخواهد پیچیدگی مدل را کاهش بدهد در واقع مجبور می‌شود از دقت مدل بکاهد. این موضوع البته در همه الگوریتم‌ها وجود دارد ولی در RBF خیلی ملموس‌تر و و مشهودتر است.

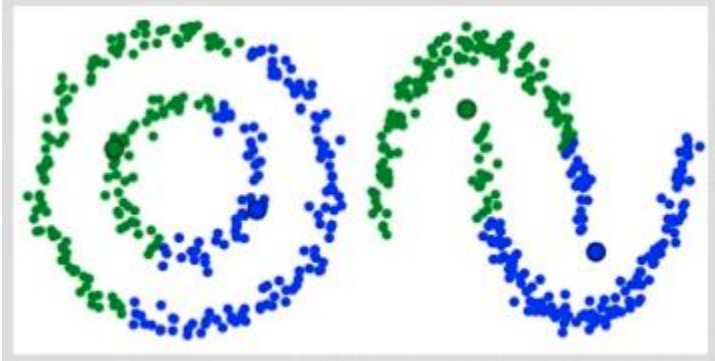
یک شبکه تابع شعاعی(RBF) ، نظیر MLP ، یک شبکه یادگیری نظارت شده است و از جهاتی به آن شباهت دارد. اما شبکه RBF فقط با یک لایه مخفی کار می کند. این کار با محاسبه مقدار هر واحد در لایه مخفی برای یک مشاهده به انجام می رسد. در واقع به جای مجموع مقادیر وزن دار واحد های سطح قبلی، از فاصله میان این مشاهدات و مرکز این واحد استفاده می کند.

بر خلاف وزن های یک پرسپترون چند لایه ، مراکز لایه پنهان یک شبکه RBF در طول یادگیری در هر تکرار تنظیم نمی شوند.

در شبکه RBF ، نورون های مخفی ، فضا را به اشتراک می گذارند و عملا از یکدیگر مستقل هستند. این امر باعث ایجاد همگرایی سریع تر شبکه‌های RBF در مرحله یادگیری می شود، که یکی از نقاط قوت آن هاست.

شبکه های MLP و RBF دو نوع رایج از شبکه پیشخور هستند. وجوه مشترک آن ها بسیار زیاد است، تنها تفاوت در روشی است که در آن واحد های پنهان ، مقادیری را ترکیب می کنند که از لایه های قبلی شبکه می آیند MLP .ها از ضرب داخلی استفاده می کنند ، در حالی که RBFها از فاصله اقلیدسی استفاده می کنند. در روش های مرسوم برای آموزش شبکه های MLP و RBF تفاوت هایی وجود دارد. همچنین می توانیم بیشتر روش های آموزش MLP را در شبکه های RBF اعمال کنیم.

یکی از اشکالات عمده الگوریتم K-Means استفاده ساده از مقدار متوسط برای مرکز خوشه است. با دیدن تصویر زیر می فهمیم که چرا این بهترین روش برای انجام کارها نیست. در سمت چپ ، برای چشم انسان کاملاً واضح به نظر می رسد که دو خوشه دایره ای با شعاع متفاوت با مرکزیت یکسان وجود دارد K-Means .نمی تواند از عهده این کار برآید زیرا مقادیر میانگین خوشه ها بسیار نزدیک به هم هستند-K . Meansهمچنین در مواردی که خوشه ها دایره ای نیستند دچار مشکل می شود.



مدل های GMM انعطاف پذیری بیشتری نسبت به K-Means به ما می دهند. با GMM ها فرض می کنیم که نقاط داده توزیع گاوسی (توزیع نرمال)هستند. با این فرض ما کمتر از قبل محدود می شویم. به این ترتیب ، ما دو پارامتر برای توصیف شکل خوشه ها داریم: میانگین و انحراف معیار. با مثالی در دو بعد ، این بدان معناست که خوشه ها می توانند هر نوع شکل بیضوی داشته باشند) از آنجا که در هر دو جهت x و y انحراف معیار داریم .(بنابراین، هر توزیع گاوسی به یک خوشه منفرد اختصاص یافته است.

برای یافتن پارامترهای Gaussian برای هر خوشه (به عنوان مثال میانگین و انحراف معیار) ، ما از یک الگوریتم بهینه سازی به نام Expectation – Maximization (EM)استفاده خواهیم کرد. به عنوان تصویری از اتصال گوسی ها به خوشه ها ، به شکل زیر نگاه کنید.

سپس می توانیم روند خوشه بندی Expectation – Maximization را با استفاده از GMM ها ادامه دهیم.

۱ -ما با انتخاب تعداد خوشه) مانند (K-Means و مقداردهی تصادفی پارامترهای توزیع گاوسی برای هر خوشه شروع می کنیم. با نگاهی گذرا به داده ها می توان پارامترهای اولیه را مقداردهی کرد ، اگرچه همانطور که در نمودار بالا مشاهده می شود ، این کار 100٪ ضروری نیست زیرا Gaussians کار ما را بسیار ضعیف آغاز می کند اما به سرعت بهینه می شود.

۲-با توجه به این که هر خوشه دارای توزیع گاوسی است ، احتمال متعلق بودن هر داده به یک خوشه خاص را محاسبه کنید. هرچه یک نقطه به مرکز Gaussian نزدیکتر باشد ، احتمال تعلق آن به آن خوشه بیشتر است. زیرا با توزیع گوسی فرض می کنیم که بیشتر داده ها به مرکز خوشه نزدیکتر هستند.

۳ -بر اساس این احتمالات ، ما پارامترهای توزیع گاوسی را محاسبه می کنیم به گونه ای که احتمالات نقاط داده را در خوشه ها به حداکثر برسانیم. ما این پارامترهای جدید را با استفاده از مجموع وزنی از موقعیت های نقطه داده محاسبه می کنیم که وزن، احتمال نقطه داده مربوط به آن خوشه خاص است. برای توضیح از نظر بصری ، می توانیم به گرافیک بالا ، به ویژه خوشه زرد ، نگاهی بیندازیم. توزیع به صورت تصادفی در اولین تکرار آغاز می شود ، اما می توان دریافت که بیشتر نقاط زرد در سمت راست آن توزیع قرار دارند. وقتی ما مجموع وزنی را با توجه به احتمالات محاسبه می کنیم ، حتی اگر در نزدیکی مرکز چند نقطه وجود داشته باشد ، بیشتر آنها در سمت راست قرار دارند. بنابراین به طور طبیعی میانگین توزیع به آن طرف نزدیکتر می شود. همچنین می توانیم ببینیم که بیشتر نقاط "بالا راست به پایین چپ" هستند. بنابراین انحراف معیار تغییر می کند و بیضی ایجاد می کند ، تا مجموع وزنی را به حداکثر برساند.

۴ -مراحل 2 و 3 تا زمان همگرایی تکرار می شود ، تا جایی که توزیع ها از به هنگام تکرار تغییر چندانی نکنند.

استفاده از GMM ها ۲ مزیت اصلی دارد. اولاً GMM ها از نظر کوواریانس خوشه ای بسیار انعطاف پذیر تر از K-Means هستند. با توجه به پارامتر انحراف استاندارد ، خوشه ها می توانند به غیر از محدود شدن به دایره ها، هر شکل بیضی به خود بگیرند K-Means .در حقیقت یک مورد خاص از GMM است که در آن کوواریانس هر خوشه در تمام ابعاد به 0 می رسد. ثانیا، از آنجا که GMM ها از احتمالات استفاده می کنند ، می توانند خوشه های مختلفی را برای هر نقطه داده داشته باشند. بنابراین اگر یک نقطه داده در وسط دو خوشه با هم تداخل داشته باشد ، می توانیم خوشه آن را به راحتی تعریف کنیم و بگوییم که آن X درصد به کلاس 1 و Y درصد به کلاس 2 است. بنابراین GMM ها از عضویت مختلط پشتیبانی می کنند.

منبع:

https://raahbord.com/perceptron-neural-network/

https://raahbord.com/k-means/

https://blog.faradars.org/neural-networks-qa-podcast/

https://shahaab-co.com/mag/edu/ml/artificial-neural-network-model/

https://deeptip.ir/5-clustering-algorithms-that-data-science-professionals-need-to-know/#khwshh_bndy_Expectation-Maximization_EM_ba_astfadh_azmdl_Gaussian_GMM_Mixture_Models

سؤال ۲: فرض: ورودی $x_1, x_2, \dots, x_n$ قابل ذخیره کردن هستند و مینیمم های محلی شبکه ی هاپفیلد دقیقاً همین ورودی هاست.

آیا لیست [(1,1,1,1), (-1,-1,-1,-1), (-1,-1,1,1), (1,1,-1,-1)] قابل ذخیره سازی است؟

۱- محاسبه ی ماتریس وزن پترن ها با توجه به قوانین شبکه هاپفیلد

← چون هر پترن لیست ۴ تایی است و ۴ پترن هم داریم فقط برای پترن ۴ عضوی حساب می کنیم و $K=4$

$$\begin{cases} w_{ij} = \sum_{K=1}^{4} w_{ij}^{4} = \sum_{k=1}^{4} x_i^k x_j^k \\ w_{ii} = 0 \\ w_{ij} = w_{ji} \end{cases}$$

0	4	0	0
4	0	0	0
0	0	0	4
0	0	4	0

حال پایداری ورودی جدید در شبکه را حساب می کنیم:

$$\begin{cases} u(i, t+1) = \sum_{j=1}^{N} w_{ij}\, a(j, t) \\ a(i, t+1) = \text{sign}(u(i, t+1)) \end{cases}$$

t: زمان

برای $n=4$ حساب می کنیم و مراحل را می نویسیم چون پترن هایمان طولش بیش از ۴ نیست.

	1	2	3	4
input ($t=0$)	x_1	x_2	x_3	x_4
$t=1$	x_2	x_1	x_4	x_3
$t=2$	x_1	x_2	x_3	x_4
	⋮	⋮	⋮	⋮
$\sum_i x_i w_{ij}$	$4x_2$	$4x_1$	$4x_4$	$4x_3$
	$4x_1$	$4x_2$	$4x_3$	$4x_4$

ماتریس وزنی که بالا بدست آوردم

در هر بار ما $\sum x_i w_{ij}$ را حساب می کردیم هر کدام ضرب ۴ می گرفتند و چون علامتش مثبت است تاثیری در پاسخ نداشت (ضریب ۴ تاثیری نداره). برای همین مثلاً برای x_2 فقط $4x_2$ را نگذاشتیم تا علامت آن حساب شود.

این ورودی ها مینیمم محلی شبکه هستند پس شبکه با این وزن ها ورودی داده شده را ذخیره می کند.

۳- مسئله ی TSP:

ابتدا نوع مسئله را مشخص میکنیم. این مسئله از نوع satisfy است یعنی وابسته به اینکه گراف شهر ها و مسیر ها به چه شکل باشد میتواند پاسخ داشته باشد یا نداشته باشد. ابتدا باید این را برای گراف داده شده تشخیص بدهیم و بعد در مرحله ی بعد دنبال پاسخ یعنی مسیری که از تمام شهر ها فقط یکبار عبور کند و به شهر اول برگردد باشیم. در اینجا یال ها مسیر ها و شهر ها راس های گراف هستند.

MLP:

منبع: اسلاید

در این مسئله ما فقط میدانیم همه ی راس ها باید در پاسخ باشند و اطلاعات دیگری نداریم. نمیدانیم کدام یال میتواند در جواب نهایی باشد و کدام یک نمیتواند و به طور خلاصه برچسبی به عنوان خروجی وجود ندارد. در حالی که شبکه ی MLP راه حلی برای حل مسائل یادگیری با نظارت است و باید داده های آموزشی خروجی یا برچسب داشته باشند تا بر اساس آن وزن ها و ترشلد ها آپدیت شوند. در نتیجه این مسئله با MLP حل نخواهد شد.

SOM:

منبع: اسلاید

مدل خود سامان دهنده یا Organization Map ، برا یآموزش نیاز به داده های برچسب دار نداریم چون این شبکه خودش با توجه به شباهت و ویژگی هایی که از داده ها پیدا میکند آنها را یاد میگیرد و دسته بندی میکند. این مدل مبتنی بر یادگیری بدون نظارت است. در نتیجه برای حل این مسئله مناسب است.

ابتدا به تعداد شهر هایی که در مسئله داریم برای شبکه نورون تعریف میکنیم. از طرفی ما از هر شهر موقعیت جغرافیایی یعنی طول و عرض جغرافیایی آن را داریم، پس برای هر شهر دو ویژگی x و y را در نظر میگیریم. وزن های این لایه را در ابتدا به صورت رندوم مقدار دهی میکنیم. متصات شهر ها را در یک ماتریس به نام X میریزیم و نسبت به میانگین نرمال میکنیم و به عنوان ورودی به شبکه میدهیم.

در ابتدا با توجه به وزن های رندوم هر شهر به صورت رندوم به یکی از نود ها مپ میشود اما با گذشت چند epoch میبینیم که تناظر بین نود ها و شهر ها با توجه به فاصله ی آنها از هم طبق مسیر ها عوض میشود. در واقع شبکه طوری طراحی شده که از نود اول به نود دوم و همینطور تا نود n ام پشت سر هم برویم. برای همین در مرحله با تغییر وزن ها و پیدا کردن شهر های نزدیک تر به هم این تناظر مدام تغییر میکند تا به جواب درست برسیم.

RBF:

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

شبکه ی RBF از توابع پایه ای شعاعی به عنوان تابع فعالیت استفاده میکند. خروجی شبکه در نهایت یک ترکیب خطی از توابع شعاعی برای پارامترهای ورودی نورون ها است.معمولا سه لایه ی ورودی ، مخفی با تابع فعالیت پایه ی شعاعی و لایه ی خروجی دارد.

الگوریتم این مدل معمولا دو مرحله دارد.

مرحله ی اول پیدا کردن مرکز ها این تابع های شعاعی در لایه ی مخفی هست. این مرحله میتونه به صورت رندوم یا با استفاده از روش های خوشه بندی انجام بشه و چون برچسبی نداریم برای یادگیری، (در هیچ مسئله ای مرکز ها از قبل مشخص نیست که مدل بخواد یاد بگیره و طبق اون انتخاب کنه مرکز ها رو و یه مرحله ی اضافه تر برای خود مدل هستش) این قسمت به نوعی یادگیری بدون نظارت است.

مرحله ی دوم به سادگی با یک مدل خطی با ضرایب w برای خروجی های لایه ی مخفی با توجه به تابع هدف، متناسب میشه.

با توجه به ساختار این شبکه پس میتوانیم مسئله ی TSP را با این مدل هم حل کنیم.

فقط باید بدانیم مرکز ها و شعاع های تابع شعاعی را چگونه انتخاب کنیم که به جواب درست همگرا شود. شعاع تابع RBF میتواند طول مسیری باشد که قرار است بدست بیاوریم. در نتیجه کم ترین مقدار و بیشترین مقدار را با توجه به وزن یال های گراف بدست می آوریم و تعداد خوشه ی مناسب را با توجه به تعداد شهر ها انتخاب میکینم. بعد بازه ی بدست آمده از مینیمم و ماکزیمم را با توجه به تعداد خوشه ها ، بر آن تقسیم میکنیم و وسط آن هارا پیدا کرده و به عنوان مرکز در نظر میگیریم. حالا هر تابع شعاعی میتواند بخشی از جواب سوال را برایمان پیدا کند یعنی کم ترین فاصله بین نود های اطراف خودش را بدست آورد و با اجتماع آنها جواب نهایی را بدست می آوریم.

Hopfield:

هاپفیلد یک شبکه ی حافظه ی انجمنی است که میتواند مدل ها را به صورت بایکولار در خود نگه دارد. در خود نگه داشتن با استفاده از تغییرات وزن ها اتفاق میفتد که بعد از یادگیری این شبکه به رشته ی ورودی مورد نظر همگرا میشود. و به جز آن میتواند با توجه به شروطی که برای وزن ها دارد به مواردی مثل معکوس ورودی داده شده هم همگرا شود و آن را هم تشخیص دهد.

این شبکه هم میتواند مسئله ی ما رو حل کند به این صورت که ما به تعداد مربع تعداد شهر ها به صورت صفحه ای نورون داشته باشیم و یال های بین شهر ها وزن نورون های متناظر را تعیین کند. مثلا اگر از یک شهر به شهر دیگر مسیری نباشد وزن اولیه ی بین نورون های مربوط به آنها صفر مقداردهی شود. سپس باید در هر مرحله مجموع حاصل ضرب وزن ها در ورودی ها را حساب کنیم و با توجه به ترشلد صفر و یک بودن خروجی نورون مربوط به آن را بررسی کنیم. سپس وزن ها را آپدیت میکنیم و همگرا شدن شبکه و مینیمم شدن تابع انرژی را چک میکنیم. مینیمم شبکه همان حالت جواب ما است. در نهایت در ماتریس وزن نهایی وزن هایی که بزرگتر از صفر باشند در مسیر پیدا شده وجود دارند.

منبع: http://azadproject.ir/wp-content/uploads/2014/12/Solving-the-Travelling-Salesman-Problem-with-a-Hopfield-type-neural-network.pdf

پیاده سازی Kohonen: کدهای مربوط به این سوال در فایل Q3.ipynb ذخیره شده اند.

اول فایل داده شده را با استفاده از کتابخانه ی pandas میخوانیم. اطلاعات ۱۹۳ شهر در فایل وجود دارد.

1) import cities file

```
import pandas as pd
df = pd.read_csv('Cities.csv',sep=" ", names = ["x", "y"])
print (df)
```

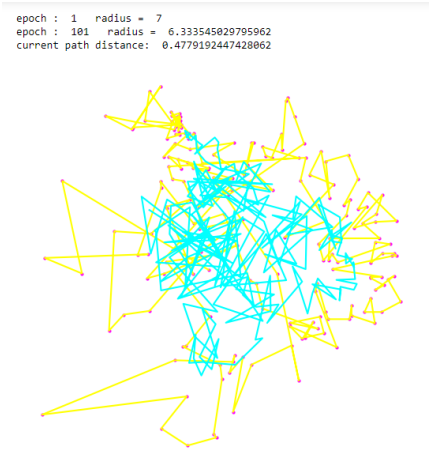
```
             x          y
1    24748.3333  50840.0000
2    24758.8889  51211.9444
3    24827.2222  51394.7222
4    24904.4444  51175.0000
5    24996.1111  51548.8889
..          ...         ...
190  26123.6111  51169.1667
191  26123.6111  51222.7778
192  26133.3333  51216.6667
193  26133.3333  51300.0000
194  26150.2778  51108.0556

[194 rows x 2 columns]
```

سپس کلاس kohonen را پیاده سازی کردیم.

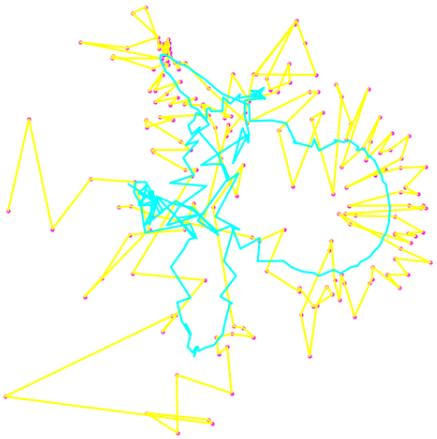
یک نمونه از آن را ساختیم و فایل داده شده را آموزش دادیم.

```
import numpy as np
import matplotlib.pyplot as plt

maximum = df.x.max()
minimum = df.x.min()
nesbat = np.array((maximum - minimum) / (maximum - minimum)) / max((maximum - minimum) / (maximum - minimum), 1)

df = df.apply(lambda c: (c - c.min()) / (c.max() - c.min()))

df = df.apply(lambda p: ratio * p, axis=1)

data = np.stack([np.array(df.x.values.tolist()), np.array(df.y.values.tolist())], axis=1)

Kohonen_Model = Kohonen(len(cities), feature_size=2, r=7, learning_rate=0.1)

pic1 = Kohonen_Model.plotting(data, fast=False)

for i in range(15):
    Kohonen_Model.train(cities_dataset, 200, decay=True)
    pic1 = Kohonen_Model.plotting(data, fast=False)
```

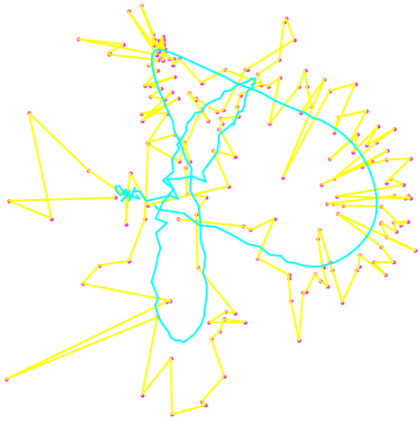
نتایج زیر حاصل شد:



```
epoch :  201   radius =  5.730541806350441
epoch :  301   radius =  5.184949225092685
current path distance:  0.028879232560455576
```



```
epoch :  401   radius =  4.691301342047169
epoch :  501   radius =  4.244652614028286
current path distance:  0.1586600929151045
```



```
epoch :  601   radius =  3.8405283523984672
epoch :  701   radius =  3.474879894017686
current path distance:  0.1586600929151045
```

```
epoch :  801   radius =  3.1440440402705208
epoch :  901   radius =  2.8447063578164236
current path distance:  0.1586600929151045
```

```
epoch :  1001   radius =  2.5738679733967404
epoch :  1101   radius =  2.3288155300368487
current path distance:  0.019921625376931474
```

```
epoch :  1201   radius =  2.107094003653789
epoch :  1301   radius =  1.9064821077363354
current path distance:  0.019921625376931474
```

همین روند ادامه پیدا کرد تا رسیدسم به نتایج زیر:

```
epoch :  2801   radius =  0.42507451969102783
epoch :  2901   radius =  0.38460408735457335
current path distance:  0.009142860494411606
```

همانطور که مشخص است کمترین طول مسیر ممکن تا تکرار ۲۹۰۱ پیدا شده است.