Computational Intelligence
Dr. Mozayani
Fall 2022
Hoorieh Sabzevari - 98412004
HW2

۱. ابتدا یک تابع برای ساخت دیتاست تعریف می‌کنیم.

از تابع آماده‌ی tarin_test_split() برای تقسیم دیتاست به دو مجموعه‌ی آموزشی و آزمایشی استفاده می‌کنیم. سپس ۴۰۰ نمونه می‌سازیم که ۰.۲ آن مجموعه‌ی تست و ۰.۸ آن مجموعه‌ی آموزشی خواهند بود.

```
def dataset_builder(n):
    x_list = []
    y_list = []
    for i in range(n):
        x = random.uniform(0, 2)
        x_list.append(x)
        L = random.uniform(-0.8, 0.8)
        y = -1 + (2/3) * np.sin(2 * x * np.pi) + L
        y_list.append(y)

    x_train, x_test, y_train, y_test = train_test_split(x_list, y_list, test_size = 0.2)
    return (x_train, x_test, y_train, y_test)
```

هر مجموعه را به آرایه تبدیل می‌کنیم.

```
x_train, x_test, y_train, y_test = dataset_builder(400)
x_train = np.array(x_train)
x_test = np.array(x_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

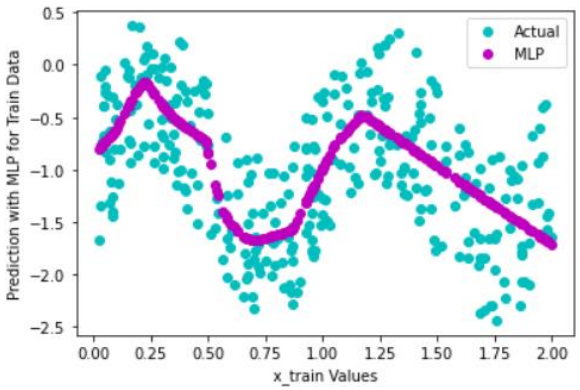
سپس مدل mlp خود را به صورت Sequential می‌سازیم. از سه لایه‌ی Dense و تابع فعالساز LeakyRelu() با آلفای ۰.۰۳ استفاده می‌کنیم. (لینک کمکی)

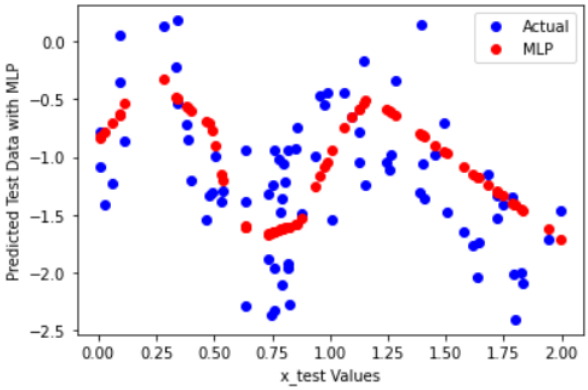
```
model = Sequential()
model.add(Dense(100, input_shape=(1,)))
model.add(LeakyReLU(alpha=0.03))
model.add(Dense(100))
model.add(LeakyReLU(alpha=0.03))
model.add(Dense(1))
```

مدل خود را با تابع ضرر MSE و تابع بهینه‌ساز Adam کامپایل و با تعداد ایپاک ۵۰۰ و batch_size ۳۲ فیت می‌کنیم و آموزش را شروع می‌کنیم. داده‌ها را هم یک دور بر می‌زنیم.

```
model.compile(loss='mean_squared_error', optimizer='Adam', metrics=['accuracy'])
history = model.fit(
    x_train, y_train,
    epochs=500,
    batch_size=32,
    shuffle=True,
    verbose=1)
```

سپس نتایج بدست‌آمده از مدل را برای ۲ مجموعه‌ی آموزشی و آزمایشی پلات می‌کنیم. همانطور که می‌بینیم نتایج مطلوبی بدست نیامده است.





حال به سراغ شبکه‌ی RBF می‌رویم.

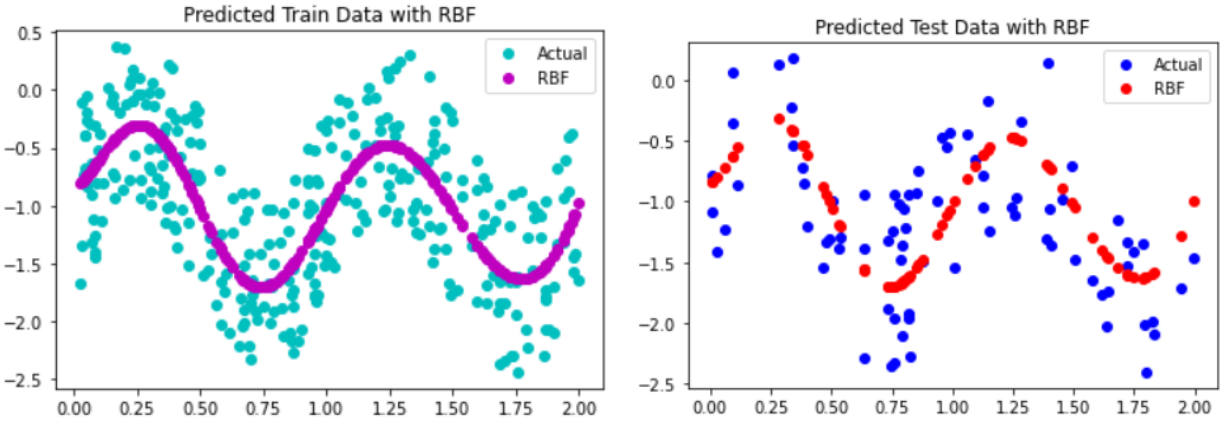
کلاس RBFN را می‌سازیم. (لینک کمکی)

مواردی که برای ساخت یک شبکه‌ی RBF نیاز داریم، تعداد نورون‌های لایه‌ی میانی، وزن‌ها و مراکز است. برای مقداردهی وزن‌ها طبق صورت سوال از سه روش استفاده می‌شود.

در این کلاس تابع _kernel_function() همان تابع شعاعی ماست که با داشتن مراکز و خود نقطه، مقدار آن را حساب می‌کند.

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 می‌دهد. حالا باید با مقایسه با مقدار y و مقدار واقعی آن ببینیم که مقدار خطای ما به چه میزان است. با داشتن میزان خطا، نرخ یادگیری و هم‌چنین مقادیر توابع شعاعی می‌توانیم تغییرات وزن را به دست آوریم.

نتایج بدست آمده از این مدل به صورت زیر است.



همانطور که می‌بینیم بهتر از مدل قبلی عمل کرده است.

به طور کلی داریم:

MLP: از ضرب نقطه‌ای (بین ورودی ها و وزن ها) و توابع فعال سازی سیگموئیدی (یا سایر توابع یکنواخت مانند ReLU) استفاده می کند و آموزش معمولاً از طریق پس‌انتشار برای همه‌ی لایه‌ها (که می‌تواند به تعداد دلخواه باشد) انجام می‌شود. این نوع شبکه عصبی در یادگیری عمیق با کمک بسیاری از تکنیک‌ها مانند dropout استفاده می‌شود.

RBF: از فواصل اقلیدسی (بین ورودی‌ها و وزن‌ها، که می‌توانند به عنوان مرکز مشاهده شوند) و (معمولا) توابع فعال سازی گاوسی (که می‌توانند چند متغیره باشند) استفاده می‌کند، که نورون‌ها را به صورت محلی حساس‌تر می‌کند. بنابراین، نورون‌های RBF زمانی که مرکز/وزن‌ها با ورودی‌ها

برابر باشند، حداکثر فعال‌سازی را دارند. با توجه به این ویژگی، شبکه‌های عصبی RBF برای تشخیص novelty خوب هستند (اگر هر نورون بر روی یک مثال آموزشی متمرکز شود، ورودی های دور از همه نورون ها الگوهای جدیدی را تشکیل می‌دهند) اما در برون‌یابی چندان خوب نیستند. همچنین، RBF ها ممکن است از پس انتشار برای یادگیری یا رویکردهای ترکیبی با یادگیری بدون نظارت در لایه پنهان استفاده کنند (آنها معمولا فقط ۱ لایه پنهان دارند). در نهایت، RBF ها رشد نورون های جدید را در طول آموزش آسان‌تر می‌کنند. به طور خلاصه، RBF ها از فاصله بین مراکز RBF و نمونه ها به عنوان معیار تشابه استفاده می کنند. یکی از معایب این است که RBF ها وزن یکسانی را به هر ویژگی می دهند زیرا در محاسبه فاصله به طور مساوی در نظر گرفته می شوند.

ما می‌توانیم نقاط کمتری را برای اجرای شبکه RBF انتخاب کنیم. همچنین می‌توانیم دلیل انتخاب امتیاز کمتر را به صورت زیر درک کنیم. اگر x1 شبیه x2 باشد، به هر دو RBF(x,x1) و RBF(x,x2) در شبکه RBF نیاز نداریم.

ما فقط می توانیم x1 و x2 را با یک نمونه اولیه u خوشه بندی کنیم که u شبیه x1 و x2 است.

برای خوشه بندی، باید کارهای زیر را انجام دهیم:

همه نقاط را به مجموعه های K جدا تقسیم کنیم. $(S_1, S_2, S_3, \dots, S_k)$

نمونه اولیه u را برای هر مجموعه انتخاب کنیم.

K-Means:

(لینک کمکی)

در این الگوریتم با توجه به تعداد دسته‌هایی که در ابتدا مشخص می‌کنیم تعدادی مرکز درنظر می‌گیریم. سپس برای هر داده آن را به دسته با نزدیک‌ترین مرکز نسبت می‌دهیم و در نهایت داده‌های با مرکز مشترک تشکیل یک دسته می‌دهند. مراحل آن به شرح زیر است:

۱. در ابتدا باید با توجه به تعداد مرکز به صورت تصادفی هر مرکز را مقدار دهی اولیه کنیم.

۲. با توجه به مکان مراکز و مقادیر داده‌های ورودی برای هر داده نزدیک‌ترین مرکز را انتخاب می‌کنیم.

۳. با توجه به مرحله‌ی ۲، میانگین داده‌هایی که به هر مرکز نگاشت شده‌اند را می‌یابیم.

۴. مقدار جدید هر مرکز را بر اساس مقدار میانگین بدست‌آمده در مرحله‌ی۳ بروزرسانی می‌کنیم و دوباره به مرحله ۲ می‌رویم.

این مراحل را تا زمانی که مقادیر هر مرکز به پایداری برسد، تکرار می‌کنیم.

الگوریتم k-means بسیار محبوب است و در برنامه‌های مختلفی مانند تقسیم‌بندی بازار، خوشه-بندی اسناد، تقسیم‌بندی تصویر، فشرده‌سازی تصویر و ... استفاده می‌شود.

Gaussian Mixture Models(GMM):

(لینک کمکی)

مدل‌های مخلوط گاوسی یک مدل احتمالی برای نشان دادن زیرجمعیت‌های معمولی توزیع شده در یک جمعیت کلی هستند. مدل‌های مخلوط به طور کلی نیازی به دانستن اینکه یک نقطه داده متعلق به کدام زیرجمعیت است، ندارند، و به مدل اجازه می‌دهند تا زیرجمعیت‌ها را به طور خودکار یاد بگیرند. از آنجایی که انتساب زیرجمعیت مشخص نیست، این نوعی یادگیری بدون نظارت است.

به عنوان مثال، در مدل‌سازی داده‌های قد انسان، قد معمولاً به‌عنوان توزیع نرمال برای هر جنسیت با میانگین تقریباً ۱۰'۵ اینچ برای مردان و ۵'۵ اینچ برای زنان مدل‌سازی می‌شود. تنها با توجه به داده های ارتفاع و نه تخصیص جنسیت برای هر نقطه داده، توزیع همه ارتفاعات از مجموع دو توزیع نرمال مقیاس شده (واریانس متفاوت) و تغییر یافته (میانگین متفاوت) تبعیت می‌کند. مدلی که این فرض را ایجاد می‌کند نمونه ای از مدل مخلوط گاوسی (GMM) است. GMMها برای استخراج ویژگی از داده‌های گفتاری مورد استفاده قرار گرفته‌اند، و همچنین به طور گسترده در ردیابی شی چندین اشیاء استفاده شده‌اند، که در آن تعداد اجزای مخلوط و میانگین آنها مکان‌های شی را در هر فریم در یک دنباله ویدیو پیش‌بینی می‌کند.

مراحل:

۱. برای مقداردهی اولیه‌ی مرکزها و یافتن اعضای هر دسته در ابتدا می‌توانیم به صورت تصادفی انتخاب کنیم و یا برای نتیجه‌ی بهتر ابتدا با استفاده از الگوریتم K-means مرکز هر دسته و اعضای اولیه آن را مقداردهی کنیم.

۲. بخش یادگیری برای هر دسته شامل تغییر میانگین و واریانس هر دسته می‌شود که دارای این تغییر با توجه به داده های موجود و احتمال بودن آن‌ها در این دسته صورت می‌گیرد.

۲.۱. ابتدا برای هر داده در هر دسته با توجه به توزیع فعلی آن دسته (میانگین و واریانس آن دسته) احتمال بودن این داده در دسته را می‌یابیم و مقدار P را برای آن در نظر می‌گیریم.

۲.۲. پس از یافتن این احتمالات برای تمامی داده‌ها برای بروزرسانی میانگین و واریانس وزن‌دار هر دسته با توجه به وزن هر داده، که برابر P به دست‌آمده در مرحله قبل است اقدام می‌کنیم.

۳. این مراحل را تا جایی که دسته‌ها به پایداری برسند تکرار می‌کنیم در نهایت برای هر داده می‌توانیم احتمال بودن در یه دسته را با توجه به توزیع آن دسته بیابیم.

روش سوم هم مقداردهی وزن‌ها بر اساس رندوم و تصادفی است.

۲. بله قابل ذخیره‌سازی است. در این سوال ۴ الگو شامل صفر و یک داریم، پس به ۴ نورون نیاز خواهیم داشت. طبق فرمول زیر ماتریس وزن‌ها را محاسبه می‌کنیم. $(k = 4)$

$$w_{ij} = x_i^k . x_j^k$$

P_1= [1 1 1 1]

$$w_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

P_2= [-1 -1 -1 -1]

$$w_2 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

P_3= [1 1 -1 -1]

$$w_3 = \begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix}$$

P_4= [-1 -1 1 1]

$$w_4 = \begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix}$$

حال بردار وزن را از جمع ۴ بردار بالا بدست می‌آوریم.

$$w = \sum w_{ij}^k$$

$$w = \begin{bmatrix} 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 \end{bmatrix}$$

هدف پیدا کردن کمترین مقدار برای تابع انرژی است. لذا طبق فرمول زیر مقدار انرژی را برای هر چهار الگو پیدا می‌کنیم.

$$E(o) = -\sum_{i,j} w_{i,j}\, o_i o_j$$

$$E([1\ 1\ 1\ 1]) = -(4+4+4+4) = -16$$

$$E([-1\ -1\ -1\ -1]) = -(4+4+4+4) = -16$$

$$E([1\ \ 1\ -1\ -1]) = -(4+4+4+4) = -16$$

$$E([-1\ -1\ \ 1\ \ 1]) = -(4+4+4+4) = -16$$

حداقل میزان انرژی ۱۶- است و چون هر ۴ الگو دارای مینیمم انرژی هستند پس قابل ذخیره-سازی نیز هستند.

۳. (لینک کمکی)

مسئله فروشنده دوره‌گرد یک چالش شناخته شده در علوم کامپیوتر است. این مسئله شامل یافتن کوتاه ترین مسیر ممکن است که همه شهرها را در یک نقشه معین فقط یک بار طی می‌کند و N-complete است.

MLP: حل مسئله با روش MLP امکان‌پذیر نیست، زیرا این یک مسئله‌ی unsupervised است و برای حل کردن آن باید یک لیبل برای هر شهر از قبل بدانیم اما اطلاعی از جایگاه شهرها در مسیر خود نداریم.

Hopfield: با توجه به لینک این مقاله این مسئله با شبکه‌ی هاپفیلد قابل‌حل‌شدن است. یک شبکه با تعداد n نورون می‌سازیم که n تعداد شهرهاست. سپس وزن‌های بین نورون‌ها را طبق فواصل بین آن‌ها مقداردهی اولیه می‌دهیم. در انتهای آموزش اگر وزن هر کدام از نورون‌های شبکه در جایگاه مختلف یک شود بهترین مسیر است.

RBF: با توجه به لینک این مقاله این مسئله با RBF قابل‌حل‌شدن است. ابتدا باید حداقل و حداکثر فاصله بین شهرها را پیدا کنیم. سپس چند بازه را تعیین کرده و با یکی از الگوریتم‌ها مرکز و عرض آن را پیدا می‌کنیم. در هر iterval می‌توانیم مسیرهایی را که کمترین فاصله را دارند پیدا کرده و به مسیر نهایی مدنظر خود برسیم.

SOM: ازآنجا که SOM داده ها با ویژگی ها شبیه هم را در نزدیکی هم قرار می‌دهد می‌توان از این روش برای مینیمم کردن فاصله ها استفاده کرد.

برای استفاده از این شبکه برای حل TSP ،مفهوم اصلی درک نحوه تغییر تابع همسایگی است. اگر به جای یک شبکه، یک آرایه دایره‌ای از نورون ها را اعلام کنیم، هر گره فقط از نورون های جلو و پشت خود آگاه خواهد بود. یعنی شباهت درونی فقط در یک بعد کار خواهد کرد. با انجام این اصلاح جزئی، نقشه‌ی خودسازماندهی مانند یک حلقه الاستیک رفتار میکند و به شهرها نزدیک‌تر می‌شود اما به لطف عملکرد همسایگی تلاش می‌کند تا محیط آن را به حداقل برساند. اگرچه این اصلاح ایده اصلی پشت این تکنیک است، اما آن‌طور که هست کار نخواهد کرد. الگوریتم به سختی همگرا

می شود. برای اطمینان از همگرایی آن، میتوانیم نرخ یادگیری α را برای کنترل کاوش و بهره‌برداری از الگوریتم در نظر بگیریم. برای به دست آوردن اکتشاف بالا در ابتدا، و پس از آن بهره برداری بالا در اجرا، ما باید هم در تابع همسایگی و هم در نرخ یادگیری یک کاهش را لحاظ کنیم. کاهش سرعت یادگیری، جابجایی تهاجمی کمتری از نورون‌های اطراف مدل را تضمین می‌کند و تضعیف همسایگی منجر به بهره‌برداری متوسط‌تر از مینیمم محلی هر بخش از مدل می‌شود.

نقشه‌ی خود سازماندهی (یا نقشه kohonen یا SOM) نوعی شبکه عصبی مصنوعی است که همچنین از مدل‌های بیولوژیکی سیستم‌های عصبی دهه ۱۹۷۰ الهام گرفته شده است. این یک رویکرد یادگیری بدون نظارت را دنبال می‌کند و شبکه خود را از طریق یک الگوریتم یادگیری رقابتی آموزش می دهد. SOM برای تکنیک‌های خوشه‌بندی و نقشه‌برداری (یا کاهش ابعاد) برای نگاشت داده‌های چند بعدی بر روی ابعاد پایین‌تر استفاده می‌شود که به افراد اجازه می‌دهد مشکلات پیچیده را برای تفسیر آسان کاهش دهند. SOM دارای دو لایه است، یکی لایه ورودی و دیگری لایه خروجی است.

پیاده‌سازی:

(لینک کمکی)

ابتدا از روی فایل داده‌شده مختصات شهرها را استخراج کرده و در آرایه‌ای ذخیره و نرمالایز می‌کنیم.

```
file = open('Cities.csv')
my_file = csv.reader(file)
lines = []
x = []
y = []
for line in my_file:
    x_point = float(line[0].split()[1])
    y_point = float(line[0].split()[2])
    x.append(x_point)
    y.append(y_point)
    lines.append([x_point, y_point])

x_y = np.array(lines)
c = x_y.shape[0]
ratio = np.sqrt((max(x) - min(x)) * (max(x) - min(x)) + (max(y) * min(y)) * (max(y) * min(y)))
normalized_x_y = (x_y - np.array([min(x), min(y)])) / ratio
```

سپس چهار تابع کمکی، به ترتیب برای ساخت شبکه، احتساب مختصات نورون برنده، احتساب مقادیر همسایگی و پلات کردن نتایج تعریف می‌کنیم.

```
# Generate a neuron network of a given size
def som_network(size):
    return np.random.rand(size, 2)

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

# Get the range gaussian of given radix around a center index.
def get_neighborhood(center, r, domain):
    if r < 1:
      r = 1
    deltas = np.absolute(center - np.arange(domain))
    distances = np.minimum(deltas, domain - deltas)
    return np.exp(-(distances * distances) / (2 * (r * r)))

# Plot a graphical representation of the problem
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()
```

سپس آموزش را آغاز کرده و نتایج را پس از هر ۲۰۰۰ ایتریشن پلات می‌کنیم.

در نهایت این مسئله به همگرایی نهایی نرسیده است اما در صورت بهینه‌تر کردن کد خواهد رسید.