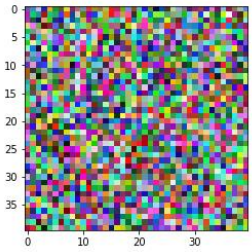
فهرست

توضیحات سوال اول:

سوال اول شامل سه قسمت بود که مورد سوم امتیازی بود. در مورد اول هدف صرفا طراحی یک مدل SOM بود که نه عملیات کاهش پارامتر learning rate دارد و نه عملیات کاهش شعاع همسایگی. در همان کلاسی که در قسمت اول ساخته شده است همه موارد هندل شده است و صرفا Decay مربوط به این موارد را عدد 1 قرار داده شده است. (که به معنی آن است که تغییری در موارد بالا با اجرای برنامه ایجاد نخواهد شد.)

برای ایجاد دیتاست رندوم از همان numpy استفاده شده است به همان تعدادی که در مسئله گفته شده است داده ساخته میشود و آبجکت های لازیم برای train شدن ساخته میشود. در هر بخش یکبار با روش Euclidean و یکبار با روش Cosine عملیات انجام شده است.




```
# generating 1600 rgb colors
pixels = np.random.randint(0, 255, NUMBER_OF_PIXELS * 3) / 100  # scale them into [0, 2.55]
pixels = pixels.reshape(NUMBER_OF_PIXELS, 3)  # reshape them to rgb shape 1 * 3
initial_pixels = pixels.reshape((40, 40, 3))
plt.imshow((initial_pixels * 100).astype(int))
plt.show()
```

مورد الف:

```
LEARNING_RATE_DECAY = 1
UPDATE_RANGE_FACTOR = 0.25
UPDATE_RANGE_DECAY = 1
```

در این مرحله دو Decay ما 1 هستند که یعنی تاثیری روی پارامترهای مورد نظر ندارند. UPDATE_RANGE_FACTOR در حقیقت بیانگر آن است که در شروع لرنینگ شعاع همسایگی را چه نسبتی از طول نقشه در نظر بگیرد به عنوان مثال اگر نقشه ما 10 در 10 باشد با توجه به عدد 0.25 شعاع همسایگی 2 در نظر گرفته میشود. اگر میتوانستیم در ابتدا شعاع را بزرگتر بگیریم و رفته رفته آن را کم کنیم خیلی نتیجه بهتری میتوانستیم بگیریم.

کلیت کلاس Kohonen به این صورت است که در constructor طول و عرض نقشه و ابعاد هر خانه داده میشود که در این جا این ورودی ها 40، 40 و 3 هستند.

```
def __init__(self, width, height, dimension):
    self.x = width
    self.y = height
    self.node_dim = dimension

    # fitting main variables
    self.n_iter = 0  # number of the iterations
    self.batch_size = 0  # number of each batch
    self.input_arr = np.array([])  # input array, given in fit function
    self.dist_method = Kohonen.NONE  # the distance mechanism

    self.node_vectors = np.array([])  # nodes' vectors

    # Learning rate
    self.learning_rate = 0

    self.update_range = 0  # the range and distance that would be updated
    self.neighbor_diff = np.array([])

    self.weights = np.array([])  # the neighbor weights
    self.x_delta = []
    self.y_delta = []
```

ورودی تابع اصلی ما که مدل را train میکند یا همان تابع fit بصورت زیر است.

```
def fit(self, input_arr, n_iter, batch_size=30, learning_rate=0.25, random_sampling=1.0, dist_method=EUCLIDEAN):
    """
    fit method is for training kohonen map.
    :param input_arr:
    :param n_iter:
    :param batch_size:
    :param learning_rate:
    :param random_sampling:
    :param dist_method:
    :return:
    """
```

مواردی تکراری مانند دیتاست ورودی، تعداد تکرار، سایز batch و همین طور learning rate هستند و فکر کنم لازم به توضیح نیست. یک random_sampling داریم که از کل دیتاست ما نسبتی را انتخاب میکند و در ادامه آن‌ها را بچ بچ میکند تا با سرعت بیشتری عملیات یادگیری تکمیل شود. همچنین dist_method بیانگر نوع سنجش فاصله هست که روش Euclidean و Cosine در این کلاس زده شده و در ابتدا تابع فیت یکی باید انتخاب بشود.

در ادامه نقشه خود را initialize میکنیم و اعدادی که جزیت میکنیم به هر حال رندوم هستند و مشکلی نباید بوجود بیاورند ولی با این حال ما میانگین مجموعه ورودی را هم در نظر میگیریم و یک ضریب کلی را در آن لحاظ میکنیم.

```
def init_map(self):
    """
    generates random numbers for node vectors and normalize them with mean of the input array.
    :return:
    """
    random_array = np.random.rand(self.y, self.x, self.node_dim)
    normal_factor = (np.mean(self.input_arr) / np.mean(random_array))
    self.node_vectors = random_array * normal_factor
```

نکته دیگری که قابل ذکر است این است که موقع آپدیت کردن هر node رو نقشه باید اطرافیان آن را آپدیت کنیم و خوب میتوانیم برای راحتی کمی این نقشه را در هنگام train و fit شدن بزرگتر فرض کرد تا درگیر جزئیات همسایگی node ها قرار نگیریم. (مثلا ممکن است یک node در گوشه باشد و سمت چپ و راست نداشته باشد و ...)

```
# add some fake element to the vectors to ease the process of updating neighbors
radius = self.update_range
size = 2 * radius
tmp_node_vectors = np.zeros((self.y + size, self.x + size, self.node_dim))
tmp_node_vectors[radius: radius + self.y, radius: radius + self.x] = self.node_vectors.copy()
self.node_vectors = tmp_node_vectors
```

در ادامه با توجه به random_sampling یک میزان از دیتاست را بر میداریم و تعداد بچ های مورد نیاز آن را حساب میکنیم.

```
# sampling and batches
n_samples = int(self.input_arr.shape[0] * random_sampling)
data_idx_arr = np.arange(self.input_arr.shape[0])
np.random.shuffle(data_idx_arr)
batch_count = math.floor(n_samples / self.batch_size)
```

در ادامه نیاز است تابع calculate_neighbor_diff را یکبار صدا داریم. همانطور که میدانید برای حساب کردن معادله تابع Normal یا Gaussian bell نیاز است معادله زیر را بدست آوریم.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

در این جا محاسبه عبارت رو به رو کمی از طرفی دردسر دارد ولی از طرفی هم برای همه یکی است و تنها یکبار لازم است مجذور اختلاف برای محاسبه واریانس یا sigma یا strandard deriviation حساب شود. در این تابع فقط یکبار اینکار با for انجام خواهد شد.

در ادامه وارد iteration اصلی میشویم که در هر حلقه تعدادی بچ با توجه به سایز batch هامون اجرا خواهند شد. در ابتدای هر iteration یک بار باید متد calculate_neighbor_effect را که ورودیش iteration است، صدا بزنیم. در حقیقت این متد همان معادله Gaussian bell را برای ما حساب میکند ولی خوب اگر ضریب Decay ما برای شعاع همسایگی 1 نبود علاوه بر محاسبه کردن معادله با توجه به iteration تاثیری نیز روی مقدار شعاع همسایگی میداشت. (در قسمت ج بیشتر بحث میشود).

```
# standard deviation
standard_deviation_factor = (size / 2) * Kohonen.UPDATE_RANGE_DECAY ** (iteration * 0.3 + 1)
standard_deviation = standard_deviation_factor * (np.sum(self.neighbor_diff) ** (1 / 2)) / size ** 2

# standard calculate the Gaussian bell plot
e_factor = 1.0 / (math.sqrt(2 * math.pi) * standard_deviation)
e_exponent_factor = (-1 / 2) * (1 / standard_deviation) ** 2
weights = e_factor * math.e ** (e_exponent_factor * self.neighbor_diff)

weights /= np.max(weights)
weights = weights.reshape(size * size, 3)
self.weights = weights
```

در ادامه برای تک تک موارد نزدیک ترین نود به آن مقدار را پیدا میکنیم و در داخل closest_nodes_idx قرار میدهیم تا با استفاده از تابع آپدیت و پاس دادن این index ها قادر به عملیات آپدیت باشیم.

```
# training
for iter_idx in range(self.n_iter):

    # calculate update effect on the neighbors
    self.calculate_neighbor_effect(iter_idx)


    total_dist = 0

    for batch in range(batch_count):
        closest_nodes_idx = np.zeros((self.batch_size, 3), dtype=np.int32)

        for item in range(self.batch_size):
            input_idx = data_idx_arr[batch * self.batch_size + item]
            input_vector = self.input_arr[input_idx]
            y, x, dist = self.find_best_matching_node(input_vector)
            closest_nodes_idx[item, 0] = y
            closest_nodes_idx[item, 1] = x
            closest_nodes_idx[item, 2] = input_idx
            total_dist += dist

        self.update_node_vectors(closest_nodes_idx)
```

برای پیدا کردن نزدیک ترین نود از متد find_best_matching_node استفاده میکنیم. که در آن با توجه به مکانیزمی که برای سنجش فاصله داریم بهترین مورد پیدا میشود و علاوه بر مکان آن فاصله را به ما بر میگرداند. همه distance ها را هم در totalذخیره میکنیم تا بتوانیم گزارشی بصورت لاگ بر اساس آن مکانیزم ارائه بدهیم.

در متد آپدیت با توجه به نزدیک ترین node ای که از قبل بدست آوردیم باید نود ها را آپدیت کنیم حال در اینجا تاثیر روی هر نود متفاوت خواهد بود و با توجه به وزنی که در معادله نرمال بدست آوردیم و learning_rate میزانی از اختلاف آن‌ها برطرف میشود و نود را آپدیت میکنیم.

```
y = closest_nodes_idx[idx, 0]
x = closest_nodes_idx[idx, 1]
inp_idx = closest_nodes_idx[idx, 2]
input_vector = self.input_arr[inp_idx]

x_range = x + self.x_delta + self.update_range
y_range = y + self.y_delta + self.update_range

previous = self.node_vectors[y_range, x_range]

update_vector = self.weights * self.learning_rate * (np.expand_dims(input_vector, axis=0) - previous)

self.node_vectors[y_range, x_range, :] += update_vector
```

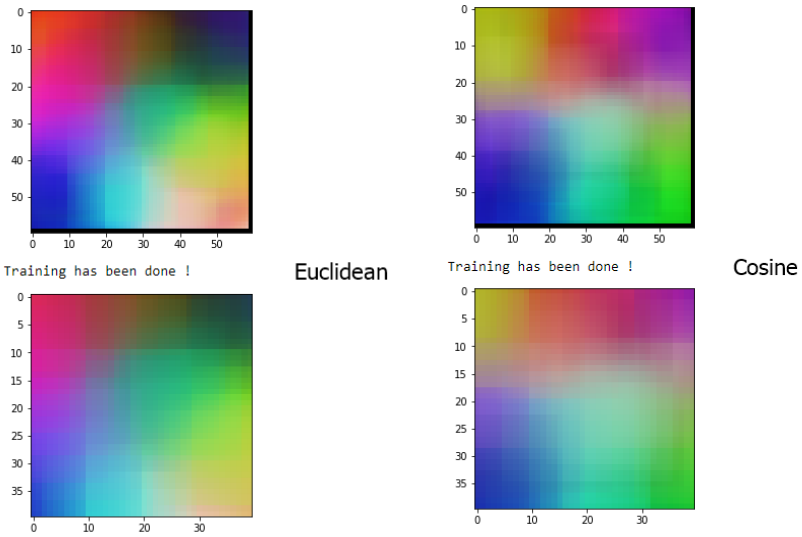
متد find distance بصورت زیر هست که صرفا معادله را پیاده سازی کردم.

```
def distance(self, vector_a, vector_b):
    """
    calculate the distance between vector_a and vector_b
    :param vector_a:
    :param vector_b:
    :return:
    """
    if self.dist_method == Kohonen.COSINE:
        # https://en.wikipedia.org/wiki/Cosine_similarity
        dist = 1. - np.dot(vector_a, vector_b) / (np.linalg.norm(vector_a) * np.linalg.norm(vector_b))
    elif self.dist_method == Kohonen.EUCLIDEAN:
        dist = np.linalg.norm(vector_a - vector_b)
    else:
        raise TypeError
    return dist
```

در انتها آن یکسری خانه های فیکی که برای آسانی خودمان اضافه کردیم را پاک میکنیم ولی من بصورت دلخواه قبل این حذف تصویری نیز از وضعیت آن نشان داده ام.

```
    import matplotlib.pyplot as plt
    plt.imshow((self.node_vectors * 100).astype(int))
    plt.show()

    # Remove padding from the vector map
    self.node_vectors = self.node_vectors[self.update_range: self.update_range + self.y,
                        self.update_range: self.update_range + self.x]
```

نتیجه مورد الف:



دو مورد نمایش داده شده است. بالایی ها هنوز خانه های fake حذف نشده اند.

مورد ب

LEARNING_RATE_DECAY = 0.76

UPDATE_RANGE_FACTOR = 0.25

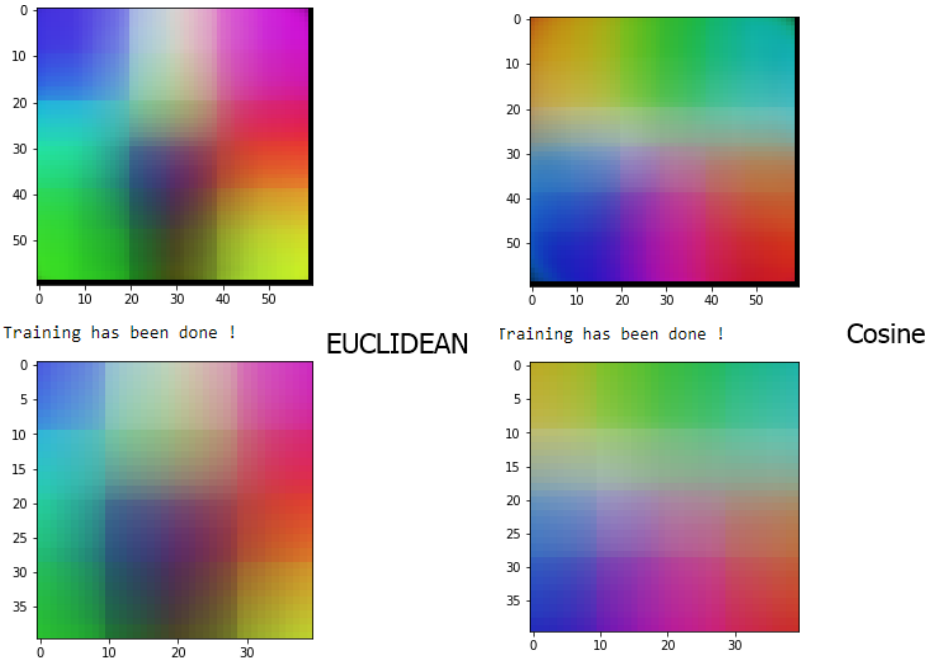
UPDATE_RANGE_DECAY = 1

این دفعه Decay را فقط برای learning_rate لحاظ کردیم. برای همین در آخر هر ایتریشن learning_rate بصورت زیر تغییر میکند. که توضیحی بیشتر از این ندارد.

```
self.learning_rate *= Kohonen.LEARNING_RATE_DECAY
```

ثابت بودن Decay برای learning_rate ایراداتی دارد. اگر ما تغییری در سرعت یادگیری ندهیم در آخرین iteration دقیقا با همان سرعت ابتدای برنامه مدل یادگیری را انجام میدهد و این باعث میشود که خوب این باعث میشود به یک پایان نرسیم و ممکن است دوباره یادگیری ما خراب شود.

نتیجه مورد ب



با استفاده از Decay که برای learning_rate گذاشتیم خیالمان بابت مشکلی که بود کمی آسوده تر میشود ولی در حالت سمت چپ همانطور که مشاهده میشود احساس میشود که بخاطر ثابت بودن شعاع مشکلاتی پیش آمده است.

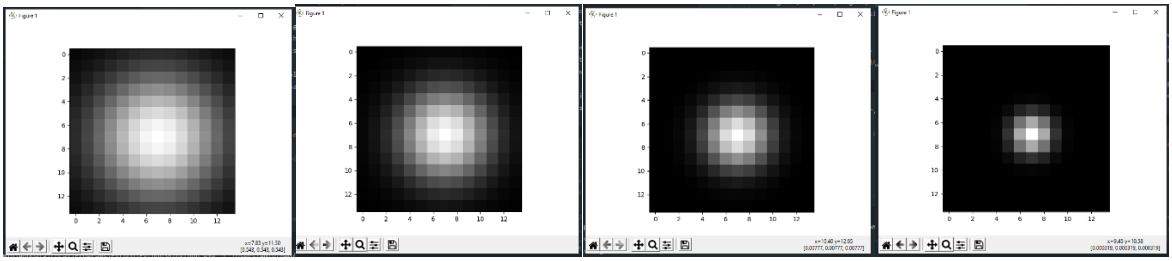
مورد ج

ثابت بودن Decay برای شعاع همسایگی که در این جا update_range نام گذاری شده است ایراد دارد که همان حالت پیچ خوردگی و پروانه ای هست. یعنی اگر این حالت بد پیش بیاید با توجه به ثابت بودن شعاع همسایگی هیچگونه نمیشه از آن جلوگیری کرد. ولی اگر در ابتدا شعاع را خیلی بزرگ در نظر بگیریم و رفته رفته آن را کم کنیم باعث میشود که اگر همچین مشکلی پیش آمده باشد برطرف شود.

حال برای این که این Decay را لحاظ کنیم باید آن را بصورت زیر لحاظ کنیم.

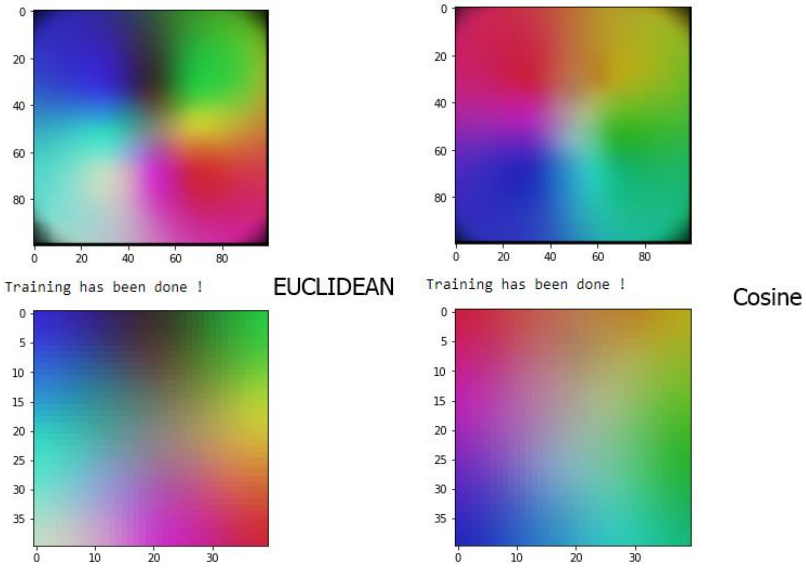
```
# standard deviation
standard_deviation_factor = (size / 2) * Kohonen.UPDATE_RANGE_DECAY ** (iteration * 0.3 + 1)
standard_deviation = standard_deviation_factor * (np.sum(self.neighbor_diff) ** (1 / 2)) / size ** 2
```

یک ضریب میسازیم که تابعی از iteration و مقدارDecay است و آن را در standard deviation یا همان سیگما ضرب میکنیم. نتیجه را مشاهده میکنیم تا ببینیم وزن‌ها چه شکلی میشوند.



در شکل های بالا میزان تاثیر گذاری روی همسایه‌ها یا همان شعاع همسایگی قابل مشاهده است که در طی iteration های یک تا سی پی در پی کوچکتر میشود و با این وجو دیگر حالت پروانه بوجود نخواهد آمد.

نتیجه مورد ج



نتیجه نهایی برای مدل Kohonen 😊

توضیحات سوال دو

در قسنا اول با استفاده از MLP تابع سینوس را یادگیری میکنیم و در قسمت دوم همینکار را با استفاده از RBF انجام خواهیم داد و در انتها آن‌ها را با هم مقایسه خواهیم کرد.

قسمت الف

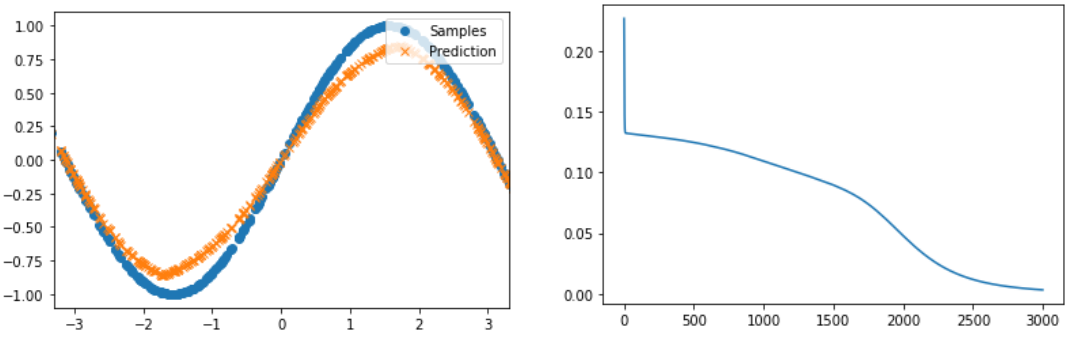
```
model = Sequential([
    Dense(units=30, input_shape=(1,), activation="relu"),
    Dense(units=60, activation="sigmoid"),
    Dense(units=1, activation="sigmoid")
])
model.summary()
```

برای مدل MLP سه لایه در نظر میگیرم که اولی 30 نرون داره دومی 60 نرون و لایه اخر هم یک نرون کافی است و تابع activation را برای این دو لایه آخر sigmoid در نظر گرفته شده است. در انتها هم بروی دیتاست تست predict صورت میگیرد.

```
batch_size = 200
epochs = 3000
# model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model.compile(loss='mean_squared_error', optimizer='SGD', metrics=['mean_squared_error', 'accuracy'])
# history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1, verbose=2)

history = History()
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1, verbose=2, callbacks=[history])
predictions = model.predict(x_test)
print(np.mean(np.square(predictions - y_test)))
```

نتیجه مورد الف

```
Test loss: 0.0034994028974324465
```



همانطور که مشاهده میکنید روی داده تستی هم بخوبی predict جواب داده و نمودار sample های ما و prediction بصورت بالا شده است.

قسمت ب

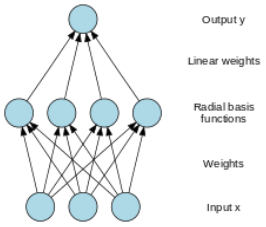
هنگام ساختن آبجکت باید تابعی که میخواید ترین بشه رو بدیم به constructor. در ادامه خود کلاس دیتاست سمپل همراه با noise تولید میکنه و تابع fit را در ادامه ما صدا میزنیم.

```
def __init__(self, func):
  self.func = func
```

این دو تابع دیتاست تولید میکنند و در صورتی که noise داشته باشیم noise را ایجاد و به مقادیر اضافه میکند.

```
def add_noise_to_dataset(self, value, input, output):
    x_noise = value[0]
    y_noise = value[1]

    n = self.n_samples
    noise_x = np.random.uniform(-x_noise, x_noise, n).reshape(n, -1)
    noise_y = np.random.uniform(-y_noise, y_noise, n).reshape(n, -1)

    input += noise_x
    output += noise_y

    return input, output

def generate_samples(self, n_samples, noise=None):
    self.n_samples = n_samples

    x = np.linspace(-RANGE, RANGE, self.n_samples).reshape(-1,1)
    y = self.func(x)

    if noise is not None:
      x, y = self.add_noise_to_dataset(noise, x, y)

    return x, y
```

بشکل زیر مدل مربوط را ایجاد میکنیم.



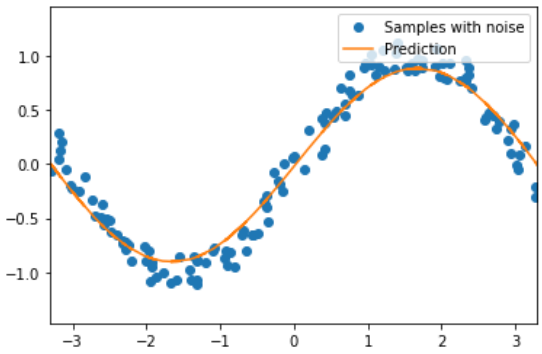

```
xx = tf.placeholder(tf.float32, [None, n_feature])
yy = tf.placeholder(tf.float32, [None, 1])

z = self.define_radial_base(xx, cluster_center, standard_dev, hidden_size, n_feature)
```

در تابع define_radial_base از همان معادله Gaussian bell استفاده شده است.

نتیجه مورد ب

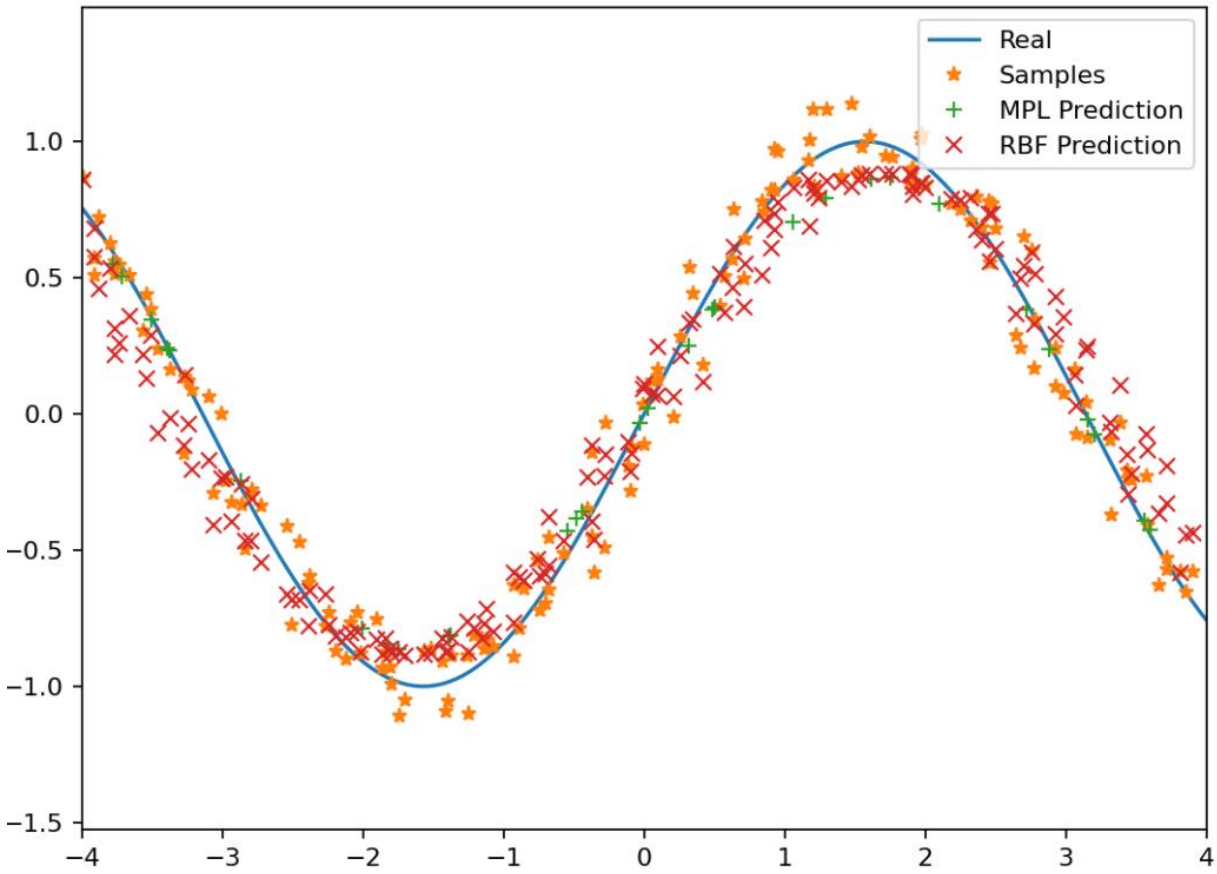
عملیات learn کردن بخوبی اجرا میشود و دیتاست noise دار با error دو صدم یادگیری را انجام میدهد.

```
iteration 10251/15001           [=========================================>                ]        Error 0.03356252983212471
iteration 10501/15001           [=========================================>                ]        Error 0.03134661912918091
iteration 10751/15001           [=========================================>                ]        Error 0.029576992616057396
iteration 11001/15001           [=========================================>                ]        Error 0.02820178121328354
iteration 11251/15001           [=========================================>                ]        Error 0.027166828513145447
iteration 11501/15001           [=========================================>                ]        Error 0.026416806504130363
iteration 11751/15001           [=========================================>                ]        Error 0.025896988809108734
iteration 12001/15001           [=========================================>                ]        Error 0.025555133819580078
iteration 12251/15001           [=========================================>                ]        Error 0.025343770161271095
iteration 12501/15001           [=========================================>                ]        Error 0.02522222511470318
iteration 12751/15001           [=========================================>                ]        Error 0.025157971307635307
iteration 13001/15001           [=========================================>                ]        Error 0.025127161294221878
iteration 13251/15001           [=========================================>                ]        Error 0.02511395514011383
iteration 13501/15001           [=========================================>                ]        Error 0.025108983740210533
iteration 13751/15001           [=========================================>                ]        Error 0.02510736882686615
iteration 14001/15001           [=========================================>                ]        Error 0.025106921792030334
iteration 14251/15001           [=========================================>                ]        Error 0.025106806308031082
iteration 14501/15001           [=========================================>                ]        Error 0.025106802582740784
iteration 14751/15001           [=========================================>                ]        Error 0.025106795132160187
iteration 15001/15001           [=========================================>]        Error 0.025106796994805336
Training has been done !
```



قسمت ج

در این قسمت نمونه های سمپل را که کمی noise دارند به همراه مقادیر بدون noise یا real نمایش میدهیم و در کنار آن پیشبینی‌های حاصل از یادگیری با مدل MLP و RBF نیز گذاشته شده است.



هر دو مدل به نظر خوب می‌آیند و توانسته اند این تابع را یاد بگیرند. نکته ای که هست مدل MLP زمان نسبتا بیشتری برای یادگیری نیاز داشت و به نظر میاید کمی دقت RBF نیز بیشتر باشد. ولی در مجموع همانطور که میبینید هر دو خوب هستند.