گزارش تمرین ۲ هوش محاسباتی — علی صداقی ۹۷۵۲۱۳۷۸

سوال ۱)

- **Algorithm properties:**
  - Dynamic learning rate
  - Dynamic neighborhood radius: to avoid distortion
  - Batch training: to make training quicker
  - Zero padded map
  - Object oriented SOM
- **Algorithm parameters:**

$width$: $width\ of\ map$

$height$: $height\ of\ map$

$depth$: $depth\ of\ map$

$units$: $kohonen\ map \mid a\ weight\ vector\ of\ shape(width, height, depth)$

$unit_{xy}$: $a\ unit\ on\ width\ x\ and\ height\ y \mid weight\ of\ the\ connection\ between\ the\ unit\ x, y\ , and\ the\ input\ vector$

d: $distance\ between\ a\ unit\ and\ the\ best\ matching\ unit\ (BMU)$

β_{xy}: $neighbourhood, representing\ a\ unit\ x, y's\ distance\ from\ the\ BMU, and\ influence\ it\ has\ on\ the\ learning.$

$iterations$: $number\ of\ iterations$

i: $current\ iteration$

$batch\_size$: $number\ of\ data\ in\ each\ batch$

b: $current\ batch$

s: $current\ step\ in\ a\ batch$

α: $learning\ rate$

λ_α: $decay\ rate\ of\ learning$

m: $number\ of\ pixels$

in: $input\ vector\ of\ shape\ (m, 3)$

$radius$: $neighboorhood\ distance$

σ: $neighboorhood\ radius$

λ_σ: $decay\ rate\ of\ sigma$

- **Algorithm procedures:**

۱. ابتدا بردار وزن units را به صورت رندوم مقدار دهی اولیه می کنیم.

```
input_average = np.mean(self.input) / 0.5
random_units = np.random.rand(self.y, self.x, self.ch) * input_average
```

۲. بردار وزن units را به اندازه شعاع همسایگی Zero Pad می کنیم.

```
zero_pad_units = np.zeros((
  self.y + 2 * self.radius,
  self.x + 2 * self.radius,
  self.ch
))
```

۳. به تعداد iteration های موجود:

```
for i in range(self.iterations):
```

a. پیکسل های موجود در بردار ورودی in را shuffle می کنیم.

```
np.random.shuffle(input_indexes)
```

b. یک همسایگی متناسب با i می‌سازیم.

```
self.make_neighborhood(i)
```

```
def make_neighborhood(self, iteration):
  size = self.radius * 2
  self.weights = np.full((size * size, self.ch), 0.0)
  p1 = 1.0 / (2 * math.pi * self.sigma ** 2)
  pdiv = 2.0 * self.sigma ** 2

  y_delta = []
  x_delta = []

  for y in range(size):
    for x in range(size):
      ep = -1.0 * ((x - self.radius) ** 2.0 + (y - self.radius) ** 2.0) / pdiv
      value = p1 * math.e ** ep
      self.weights[y * size + x] = value
      y_delta.append(y - int(self.radius))
      x_delta.append(x - int(self.radius))

  self.x_delta = np.array(x_delta, dtype=np.int32)
  self.y_delta = np.array(y_delta, dtype=np.int32)

  self.weights -= self.weights[size // 2]
  self.weights[self.weights < 0] = 0
  self.weights /= np.max(self.weights)
```

c. بردار ورودی را به batch های ریز تر تبدیل می کنیم.

```
input_count = self.input.shape[0]
input_indexes = np.arange(input_count)
batch_count = math.ceil(input_count / self.batch_size)
```

d. به تعداد batch های موجود:

```
for b in range(batch_count):
```

i. برای هر پیکسل موجود در batch:

```
for s in range(steps_in_batch):
```

۱. فاصله اقلیدسی آن پیکسل و تک تک واحد های بردار وزن را محاسبه می کنیم.

```
distance = euclidean_distance(data, mu)
```

```
def euclidean_distance(u, v):
  return np.linalg.norm(u - v)
```

۲. واحدی که کمترین فاصله را با آن پیکسل دارد BMU می‌باشد.

```
bmu, min_distance = self.find_bmu(data)
```

```
def find_bmu(self, data):
  min_distance = None
  bmu_x = None
  bmu_y = None

  for y in range(self.y):
    for x in range(self.x):
      mu = self.units[
        y + self.radius,
        x + self.radius
      ]
      distance = euclidean_distance(data, mu)
      if min_distance is None or min_distance > distance:
        min_distance = distance
        bmu_x = x
        bmu_y = y

  bmu = {
    'y': bmu_y,
    'x': bmu_x
  }
  return bmu, min_distance
```

۳. کمترین فاصله آن پیکسل با واحد ها را به فاصله‌ی کلی می‌افزاییم.

```
total_distance += min_distance
```

ii. بردار وزن واحد ها را با توجه به BMU های پیدا شده آپدیت می‌کنیم.

```
self.update_units(bmu_array)
```

```
def update_units(self, bmu_array):
  for i in range(bmu_array.shape[0]):
    unit_y = bmu_array[i, 0]
    unit_x = bmu_array[i, 1]
    data_index = bmu_array[i, 2]
    data = self.input[data_index]

    old = self.units[
      unit_y + self.y_delta + self.radius,
      unit_x + self.x_delta + self.radius
    ]

    diff = (np.expand_dims(data, axis=0) - old)
    updates = self.weights * self.learning_rate * diff

    self.units[
      unit_y + self.y_delta + self.radius,
      unit_x + self.x_delta + self.radius,
      :
    ] += updates
```

e. مقدار Learning rate را می‌کاهیم.

```
self.update_learning_rate(i + 1)
```

```
def update_learning_rate(self, iteration):
  self.learning_rate = self.learning_rate_0 / (1. + self.learning_dr * iteration)
```

f. مقدار شعاع همسایگی (سیگما) را می‌کاهیم.

```
self.update_sigma(i + 1)
```

```
def update_sigma(self, iteration):
  self.sigma = self.radius / (1. + self.radius_dr * iteration)
```

۴. Zero Padding بردار وزن واحد ها را از بین می‌بریم و به عنوان جواب بر می‌گردانیم.

```
self.units = self.units[
  self.radius : self.radius + self.y,
  self.radius : self.radius + self.x
]
```

- **Hyperparameters:**

```
WIDTH = 40
HEIGHT = 40
DEPTH = 3
ITERATIONS = 30
BATCH_SIZE = 32
LEARNING_RATE = 0.25
RADIUS = min(WIDTH, HEIGHT) / 1.3
```

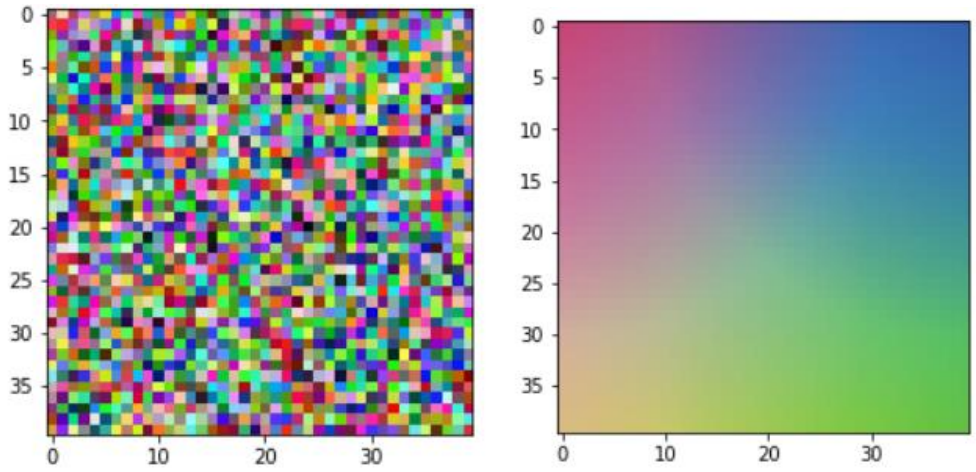
- **Result:**

در حالت اول که هیچ کاهشی بر روی نرخ یادگیری و شعاع همسایگی نداریم الگوریتم به هیچ وجه یاد نمی‌گیرد و Converge نمی-شود.

```
LEARNING_RATE_DECAY = 0
RADIUS_DECAY = 0

som.init_decay_rate(LEARNING_RATE_DECAY, RADIUS_DECAY)
som.fit(dataset, ITERATIONS, BATCH_SIZE, LEARNING_RATE, RADIUS)
train1 = som.get_units().astype(np.uint8)
```

```
Iteration: 0 Average distance: 72.26358315516126
Iteration: 1 Average distance: 73.67033786606707
Iteration: 2 Average distance: 71.60904454206059
Iteration: 3 Average distance: 72.94326997014056
Iteration: 4 Average distance: 72.6164033658663
Iteration: 5 Average distance: 71.83749163149925
Iteration: 6 Average distance: 73.1278945906211
Iteration: 7 Average distance: 73.60570590541145
Iteration: 8 Average distance: 72.93782845071487
Iteration: 9 Average distance: 72.48921426343368
Iteration: 10 Average distance: 73.41533654792272
Iteration: 11 Average distance: 73.59258415360823
Iteration: 12 Average distance: 73.43226549204671
Iteration: 13 Average distance: 73.30100920395446
Iteration: 14 Average distance: 73.58679188285994
Iteration: 15 Average distance: 72.79966940447056
Iteration: 16 Average distance: 72.43164321935753
Iteration: 17 Average distance: 73.23449709186737
Iteration: 18 Average distance: 74.00415784741303
Iteration: 19 Average distance: 74.0279910189458
Iteration: 20 Average distance: 72.21871440535769
Iteration: 21 Average distance: 71.79770494973353
Iteration: 22 Average distance: 72.87695724255448
Iteration: 23 Average distance: 73.1744943158114
Iteration: 24 Average distance: 74.04638309044266
Iteration: 25 Average distance: 74.46845797015244
Iteration: 26 Average distance: 73.4124398758612
Iteration: 27 Average distance: 72.73226250657355
Iteration: 28 Average distance: 74.39257994805999
Iteration: 29 Average distance: 73.0815178853681
```

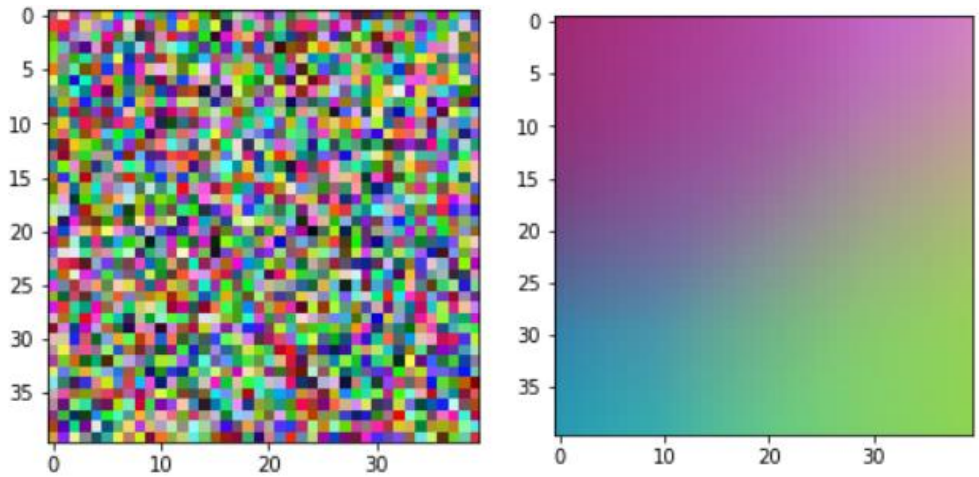


در حالت دوم که نرخ یادگیری کاهش می‌یابد اما شعاع همسایگی ثابت است نیز مشابه مورد بالا الگوریتم به هیچ وجه یاد نمی‌گیرد و Converge نمی‌شود.

```
LEARNING_RATE_DECAY = LEARNING_RATE / ITERATIONS
RADIUS_DECAY = 0

som.init_decay_rate(LEARNING_RATE_DECAY, RADIUS_DECAY)
som.fit(dataset, ITERATIONS, BATCH_SIZE, LEARNING_RATE, RADIUS)
train2 = som.get_units().astype(np.uint8)
```

```
Iteration: 0 Average distance: 72.33105801724872
Iteration: 1 Average distance: 72.49665609835083
Iteration: 2 Average distance: 72.3982811709996
Iteration: 3 Average distance: 72.4221201140159
Iteration: 4 Average distance: 72.3019547425121
Iteration: 5 Average distance: 73.46255005804343
Iteration: 6 Average distance: 72.80812151876937
Iteration: 7 Average distance: 73.04814302954826
Iteration: 8 Average distance: 72.66952928174216
Iteration: 9 Average distance: 70.9877604764599
Iteration: 10 Average distance: 73.91830842062006
Iteration: 11 Average distance: 72.54155846131125
Iteration: 12 Average distance: 71.70701443018382
Iteration: 13 Average distance: 72.14629068083705
Iteration: 14 Average distance: 72.9107112421584
Iteration: 15 Average distance: 71.39559830035786
Iteration: 16 Average distance: 72.0028160536537
Iteration: 17 Average distance: 72.7402980150264
Iteration: 18 Average distance: 73.64640829194398
Iteration: 19 Average distance: 72.72223019242736
Iteration: 20 Average distance: 72.26606874337264
Iteration: 21 Average distance: 71.0820839660683
Iteration: 22 Average distance: 73.62730663601455
Iteration: 23 Average distance: 71.83435374111912
Iteration: 24 Average distance: 73.04194797276112
Iteration: 25 Average distance: 72.36868477062954
Iteration: 26 Average distance: 73.59251045162017
Iteration: 27 Average distance: 71.74003417272336
Iteration: 28 Average distance: 71.48137457058317
Iteration: 29 Average distance: 71.4601169785536
```

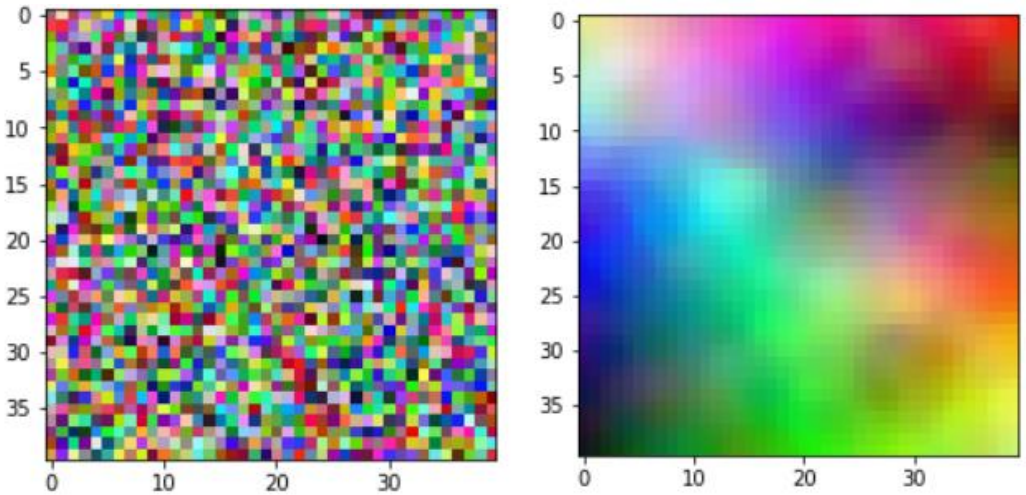


در حالت سوم که هر دو نرخ یادگیری و شعاع همسایگی کاهش می‌یابد الگوریتم بسیار سریع یاد می‌گیرد و به سرعت Converge می‌شود.

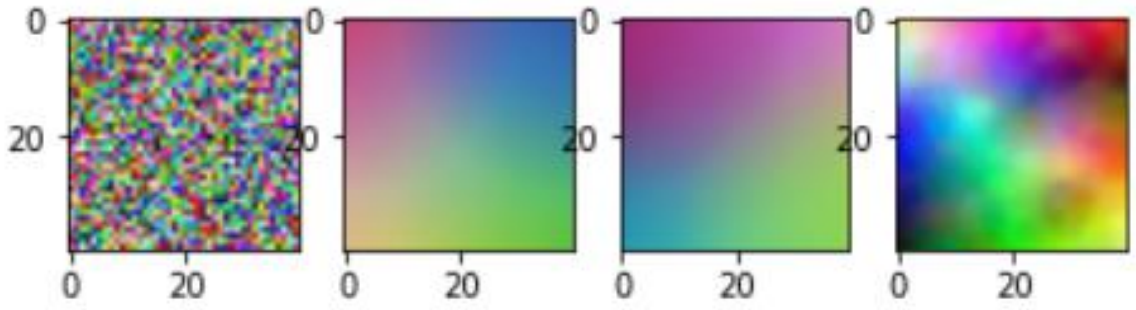
```
LEARNING_RATE_DECAY = LEARNING_RATE / ITERATIONS
RADIUS_DECAY = RADIUS / ITERATIONS

som.init_decay_rate(LEARNING_RATE_DECAY, RADIUS_DECAY)
som.fit(dataset, ITERATIONS, BATCH_SIZE, LEARNING_RATE, RADIUS)
train3 = som.get_units().astype(np.uint8)
```

```
Iteration: 0 Average distance: 72.05535729333543
Iteration: 1 Average distance: 66.36968476887924
Iteration: 2 Average distance: 56.67740232591733
Iteration: 3 Average distance: 47.961328478786946
Iteration: 4 Average distance: 39.70784454124796
Iteration: 5 Average distance: 35.01875597407359
Iteration: 6 Average distance: 31.51357985326081
Iteration: 7 Average distance: 28.723246683494825
Iteration: 8 Average distance: 26.728418434793944
Iteration: 9 Average distance: 24.63613969164833
Iteration: 10 Average distance: 23.27771294576084
Iteration: 11 Average distance: 21.93833820396544
Iteration: 12 Average distance: 20.69678985398178
Iteration: 13 Average distance: 19.527949670665123
Iteration: 14 Average distance: 18.754115994483737
Iteration: 15 Average distance: 18.200052624512605
Iteration: 16 Average distance: 17.529776981652844
Iteration: 17 Average distance: 16.927329394200004
Iteration: 18 Average distance: 16.272148439642518
Iteration: 19 Average distance: 15.871158168699585
Iteration: 20 Average distance: 15.30609122500547
Iteration: 21 Average distance: 14.80114570798733
Iteration: 22 Average distance: 14.400746761960356
Iteration: 23 Average distance: 14.038935710281308
Iteration: 24 Average distance: 13.737209652509968
Iteration: 25 Average distance: 13.361103845088728
Iteration: 26 Average distance: 12.965359446125499
Iteration: 27 Average distance: 12.664721964976382
Iteration: 28 Average distance: 12.277940377092166
Iteration: 29 Average distance: 12.043170763441529
```

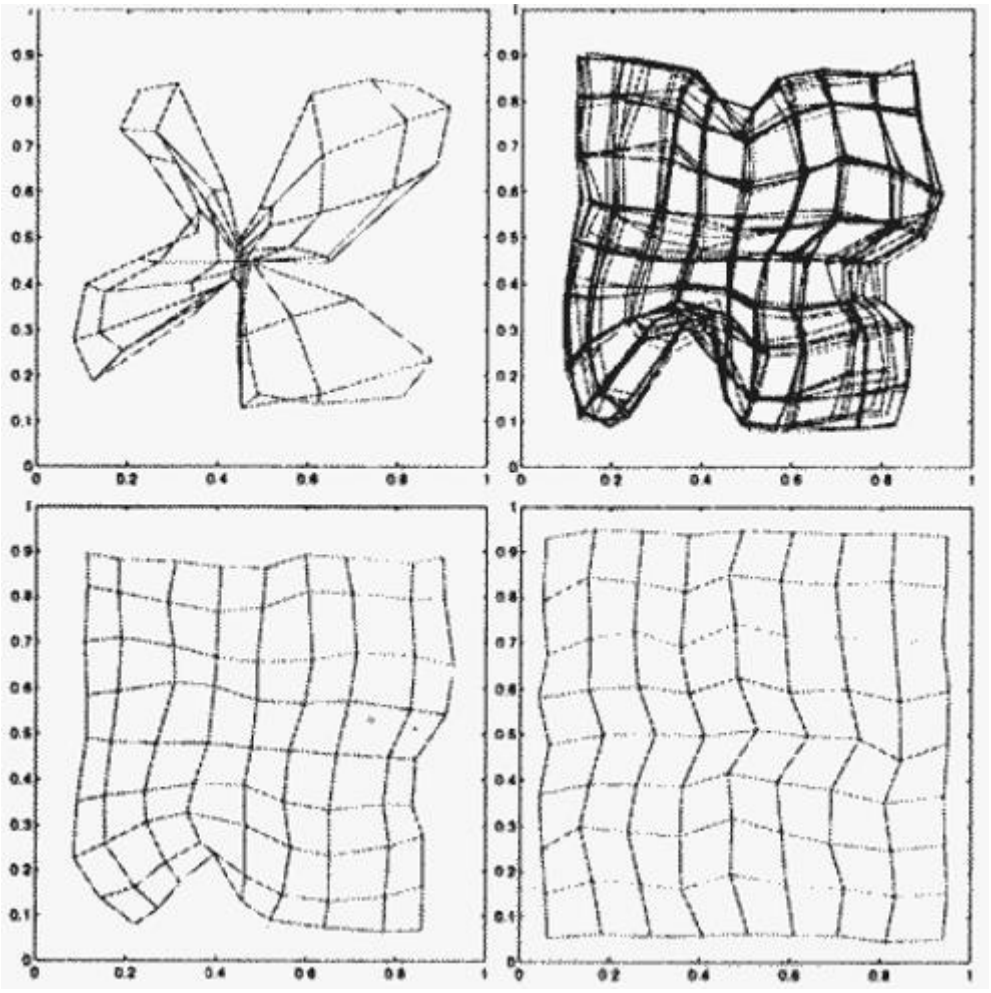


- **Compare**



ایراد ثابت بودن یادگیری: سرعت آپدیت کردن وزن ها هنگامی که الگوریتم در شروع کار (Iteration 0) می‌باشد با سرعتش در Iteration آخر یکی می‌باشد اما این کار درستی نیست چون هر چه به جلو می‌رویم وزن ها نزدیک حالت Converge می‌باشند و اگر با ضریب بالا آن ها را آپدیت کنیم ممکن است دوباره Diverge شوند.

ایراد ثابت بودن شعاع همسایگی: اگر رنگ ها در ابتدا دچار پیچ خوردگی و Distortion باشند این اثر تا آخر الگوریتم پیش می‌رود و شبکه شبیه شکل پروانه می‌شود و در مرکزش یک پیچ خوردگی بزرگ به وجود می آید. راه حل این مشکل این است که در ابتدا شعاع همسایگی بزرگ باشد و به مرور کاهش یابد. این امر باعث می‌شود که اگر در ابتدا پیچ خوردگی وجود داشت BMU با قدرت بیشتری نقشه را سمت خود بکشد و پیچ خوردگی در همان ابتدا از بین برود.



سوال ۲)

الف) با استفاده از keras یک شبکه MLP با ویژگی های زیر می‌سازیم:

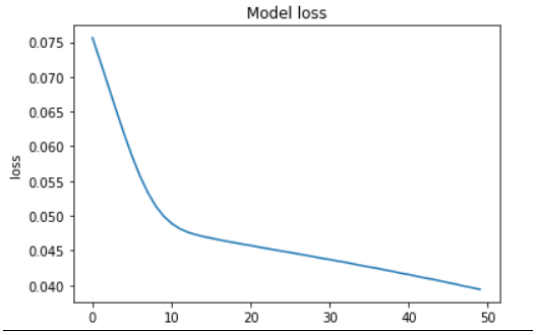
Layers: 3

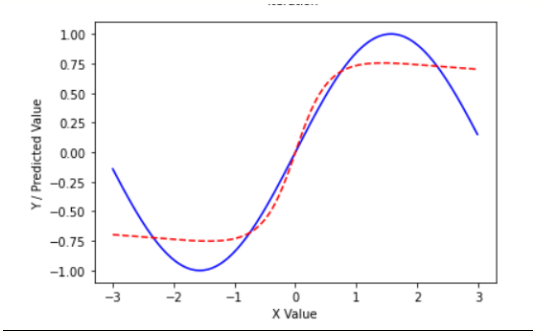
- Layer 1: Fully Connected | Number of neurons: 5 | Activation: tanh
- Layer 2: Fully Connected | Number of neurons: 5 | Activation: tanh
- Output Layer: Fully Connected | Number of neurons: 1 | Activation: tanh

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_9 (Dense)              (None, 5)                 10
_________________________________________________________________
dense_10 (Dense)             (None, 5)                 30
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 6
=================================================================
Total params: 46
Trainable params: 46
```

- ✓ از تابع فعال سازی tanh استفاده کردیم زیرا ورودی sin می‌باشد و توابع فعال سازی خطی و شبه خطی مانند ReLu نمی‌توانند آن را به خوبی یاد بگیرند.
- ✓ در لایه آخر نیز به جای Sigmoid از tanh استفاده کردیم زیرا بازه خروجی sin از -۱ تا +۱ می‌باشد در صورتی که Sigmoid عددی بین ۰ تا ۱ تولید می‌کند.
- ✓ همانطور که می‌دانیم تابع Cross Entropy Error بر روی مسائل Logistic Regression خوب عمل می‌کند اما این یک مسئله Logistic نیست پس از تابع خطا MSE استفاده کردیم.

<u>نتایج به دست آمده:</u>





ب)

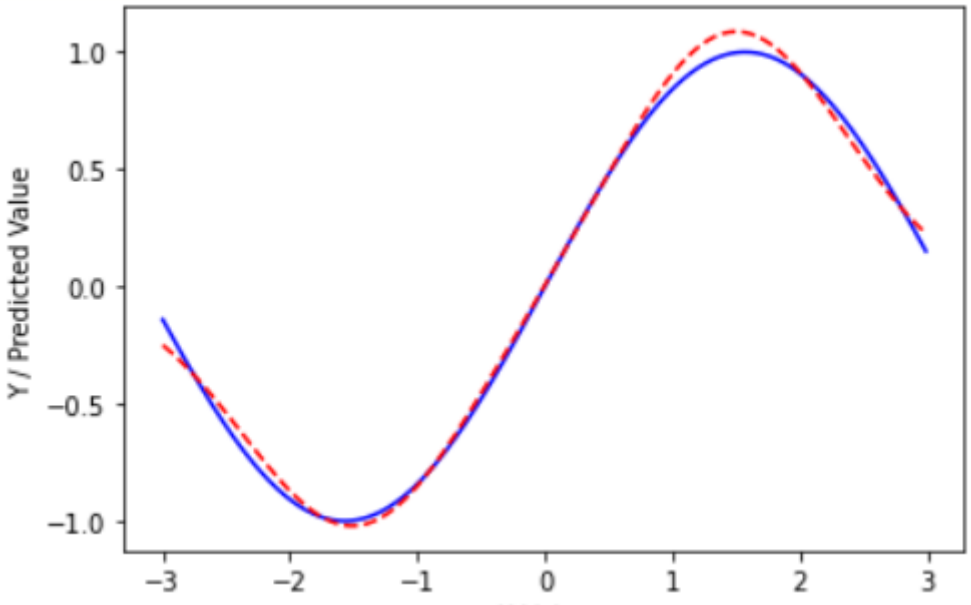
در این مدل از RBF از یک تابع که توسط فرد دیگری پیاده سازی شده برای پیدا کردن K-Means کلاستر ها استفاده شده که با ورودی گرفتن بردار ورودی مراکز و همچنین شعاع ها بر گردانده می‌شوند.

طرز train کردن به صورت Stochastic می‌باشد یعنی هر بار که یک تک نقطه دیده می‌شود پارامتر ها آپدیت می‌شوند.

هایپرپارامتر هایی که شبکه با آن ها ترین شده به صورت زیر است:

```
K = 2
ITERATIONS = 500
LEARNING_RATE = 0.01
```

نتیجه به دست آمده نسبت به حالت MLP دقیق تر و همچنین سریع تر بود.

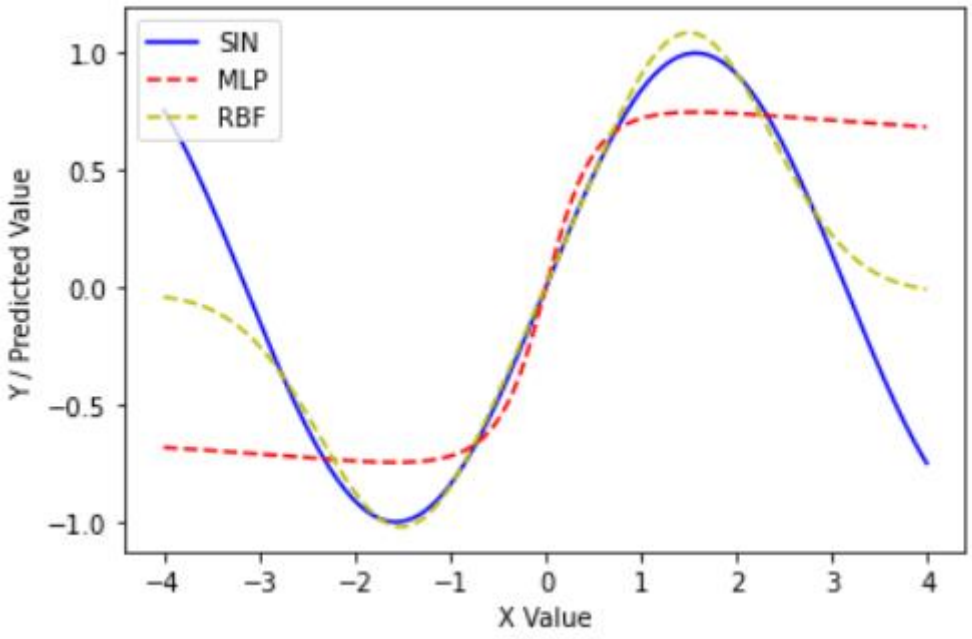


ج)

با استفاده از پارامتر های به دست آمده در مورد الف و ب شبکه را روی دیتا ست جدید که بین بازه -۴ تا +۴ می‌باشد Predict می کنیم.

نتیجه RBF بسیار دقیق تر است زیرا برای توابع غیر خطی و Classification های غیر خطی مدل های Radial عملکرد بهتری دارند به دلیل اینکه از متد های گاوسی استفاده می کنند.

نتیجه:



همانطور که مشاهده می کنید مدل RBF انطباق بیشتری روی تابع سینوس دارد.