Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ имени Н.Э.БАУМАНА
(национальный исследовательский университет)»

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа №4

«Радиально-базисная нейронная сеть»

По дисциплине «Теория искусственных нейронных сетей»

Работу выполнил

студент группы ИУ9-72Б

Владиславов А.Г.

Москва, 2021

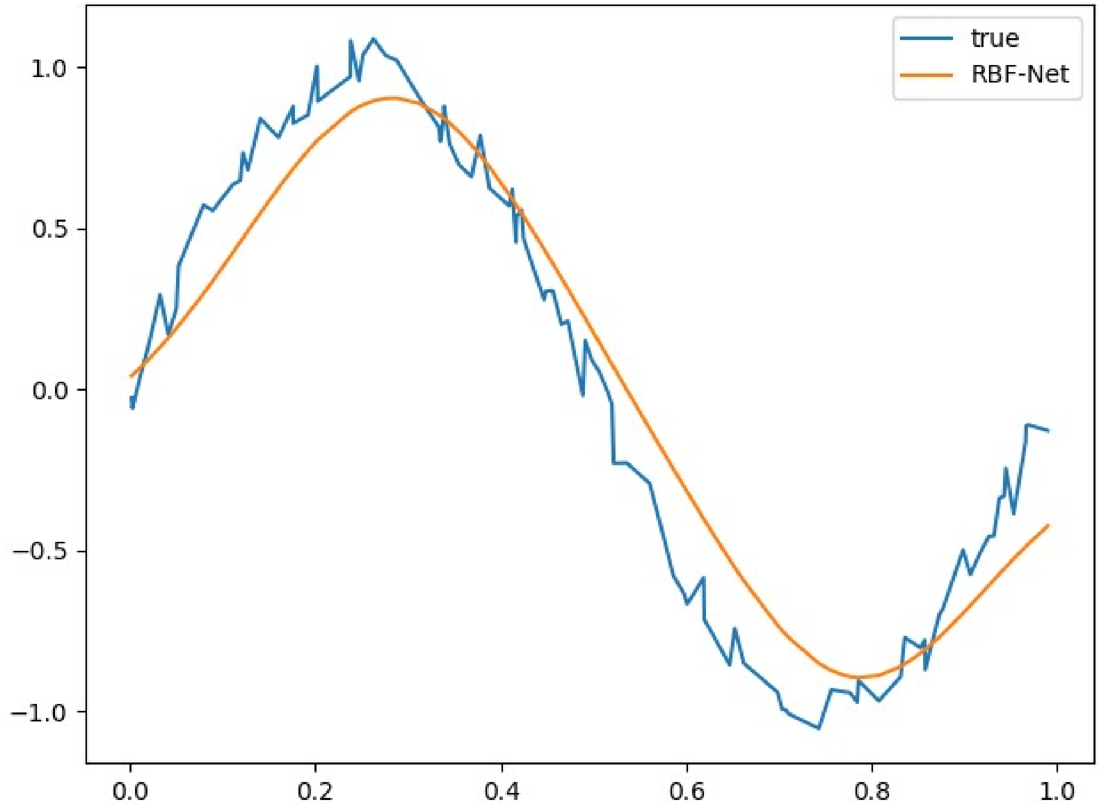
```python
import numpy as np
import matplotlib.pyplot as plt

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


def kmeans(X, k):
    """Performs k-means clustering for 1D input

    Arguments:
        X {ndarray} -- A Mx1 array of inputs
        k {int} -- Number of clusters

    Returns:
        ndarray -- A kx1 array of final cluster centers
    """

    # randomly select initial clusters from input data
    clusters = np.random.choice(np.squeeze(X), size=k)
    prev_clusters = clusters.copy()
    stds = np.zeros(k)
    converged = False

    while not converged:
        """
        compute distances for each cluster center to each point
        where (distances[i, j] represents the distance between the ith
point and jth cluster)
        """
        distances = np.squeeze(np.abs(X[:, np.newaxis] -
clusters[np.newaxis, :]))

        # find the cluster that's closest to each point
        closest_cluster = np.argmin(distances, axis=1)

        # update clusters by taking the mean of all of the points
assigned to that cluster
        for i in range(k):
            points_for_cluster = X[closest_cluster == i]
            if len(points_for_cluster) > 0:
                clusters[i] = np.mean(points_for_cluster, axis=0)

        # converge if clusters haven't moved
        converged = np.linalg.norm(clusters - prev_clusters) < 1e-6
        prev_clusters = clusters.copy()

    distances = np.squeeze(np.abs(X[:, np.newaxis] -
clusters[np.newaxis, :]))
```

```
    closest_cluster = np.argmin(distances, axis=1)

    clusters_with_no_points = []
    for i in range(k):
        points_for_cluster = X[closest_cluster == i]
        if len(points_for_cluster) < 2:
            # keep track of clusters with no points or 1 point
            clusters_with_no_points.append(i)
            continue
        else:
            stds[i] = np.std(X[closest_cluster == i])

    # if there are clusters with 0 or 1 points, take the mean std of
the other clusters
    if len(clusters_with_no_points) > 0:
        points_to_average = []
        for i in range(k):
            if i not in clusters_with_no_points:
                points_to_average.append(X[closest_cluster == i])
        points_to_average = np.concatenate(points_to_average).ravel()
        stds[clusters_with_no_points] =
np.mean(np.std(points_to_average))

    return clusters, stds


class RBFNet(object):
    """Implementation of a Radial Basis Function Network"""
    def __init__(self, k=2, lr=0.01, epochs=100, rbf=rbf,
infer_stds=True):
        self.k = k
        self.lr = lr
        self.epochs = epochs
        self.rbf = rbf
        self.infer_stds = infer_stds

        self.w = np.random.randn(k)
        self.b = np.random.randn(1)

    def fit(self, X, y):
        if self.infer_stds:
            # compute stds from data
            self.centers, self.stds = kmeans(X, self.k)
        else:
            # use a fixed std
            self.centers, _ = kmeans(X, self.k)
            d_max = max([np.abs(c1 - c2) for c1 in self.centers for c2
in self.centers])
            self.stds = np.repeat(d_max / np.sqrt(2*self.k), self.k)
```

```
        # training
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                # forward pass
                a = np.array([self.rbf(X[i], c, s) for c, s, in
zip(self.centers, self.stds)])
                F = a.T.dot(self.w) + self.b

                loss = (y[i] - F).flatten() ** 2
                # if i % 15 == 0:
                #     print('Loss {0}: {1:.2f}'.format(i, loss[0]))

                # backward pass
                error = -(y[i] - F).flatten()

                # online update
                self.w = self.w - self.lr * a * error
                self.b = self.b - self.lr * error

    def predict(self, X):
        y_pred = []
        for i in range(X.shape[0]):
            a = np.array([self.rbf(X[i], c, s) for c, s, in
zip(self.centers, self.stds)])
            F = a.T.dot(self.w) + self.b
            y_pred.append(F)
        return np.array(y_pred)


# sample inputs and add noise
NUM_SAMPLES = 100
K = 2


print("sin")
X = np.random.uniform(0., 1., NUM_SAMPLES)
X = np.sort(X, axis=0)
noise = np.random.uniform(-0.1, 0.1, NUM_SAMPLES)
y = np.sin(2 * np.pi * X) + noise

rbfnet = RBFNet(lr=1e-2, k=K, infer_stds=True)
rbfnet.fit(X, y)

y_pred = rbfnet.predict(X)

plt.plot(X, y, label='true')
plt.plot(X, y_pred, label='RBF-Net')
plt.legend()
```

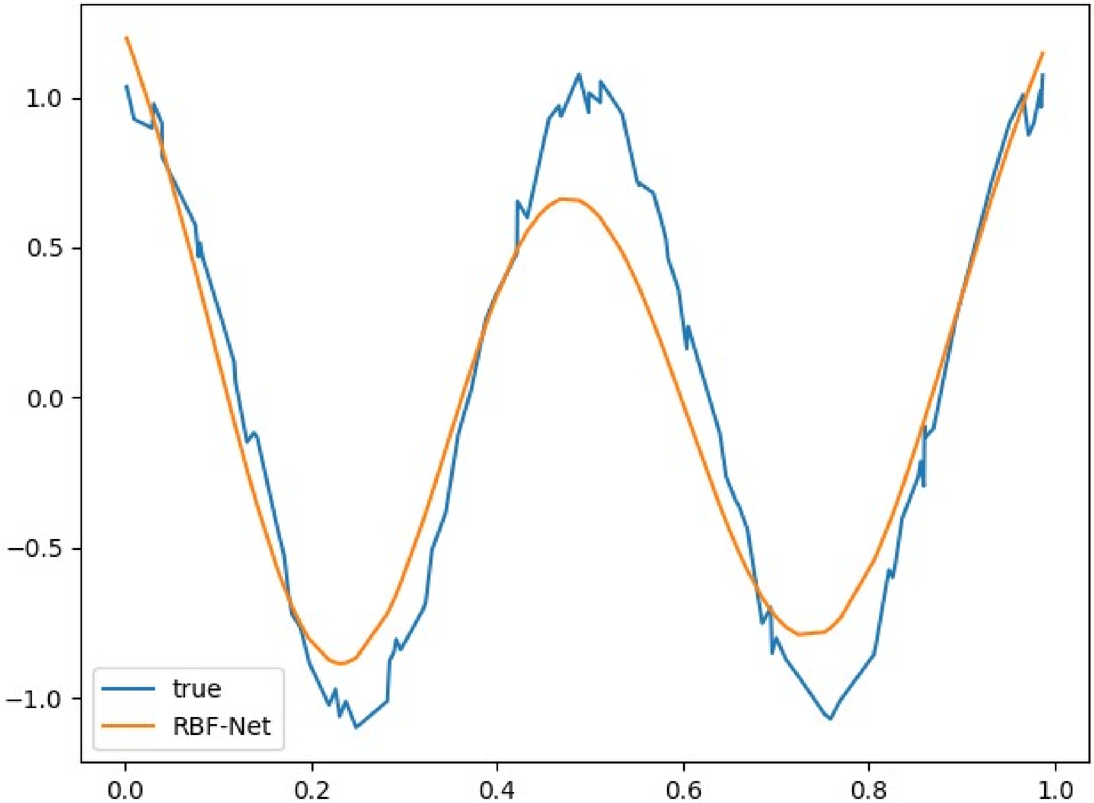
```
plt.tight_layout()
plt.show()
```

sin



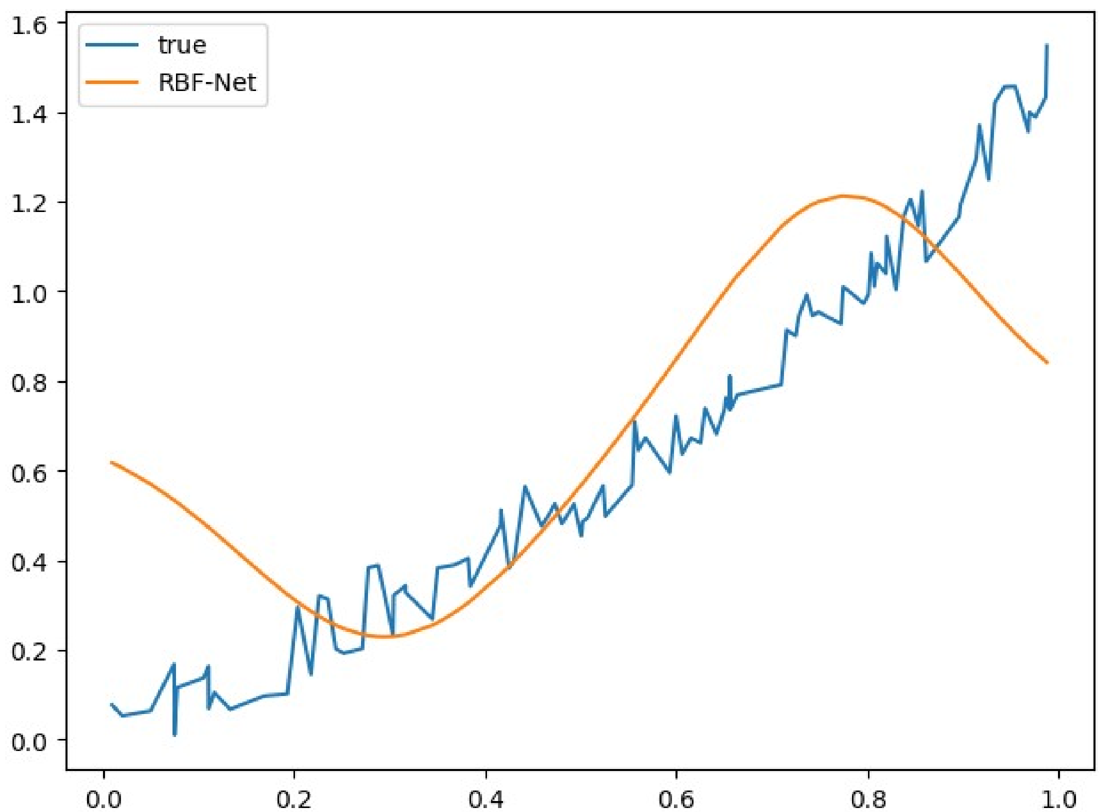

```
print("cos")
X = np.random.uniform(0., 1., NUM_SAMPLES)
X = np.sort(X, axis=0)
noise = np.random.uniform(-0.1, 0.1, NUM_SAMPLES)
y = np.cos(4 * np.pi * X) + noise

rbfnet = RBFNet(lr=1e-2, k=K, infer_stds=True)
rbfnet.fit(X, y)

y_pred = rbfnet.predict(X)

plt.plot(X, y, label='true')
plt.plot(X, y_pred, label='RBF-Net')
plt.legend()

plt.tight_layout()
plt.show()
```

cos




```
print('tg')
X = np.random.uniform(0., 1., NUM_SAMPLES)
X = np.sort(X, axis=0)
noise = np.random.uniform(-0.1, 0.1, NUM_SAMPLES)
y = np.tan(X) + noise

rbfnet = RBFNet(lr=1e-2, k=K, infer_stds=True)
rbfnet.fit(X, y)

y_pred = rbfnet.predict(X)

plt.plot(X, y, label='true')
plt.plot(X, y_pred, label='RBF-Net')
plt.legend()

plt.tight_layout()
plt.show()
```

tg


true
RBF-Net
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6