Sharif University of Technology
Electrical Engineering School

Computational Intelligence CHW2

RBF NETWORKS, MULTI LAYER PERCEPTRON, CLUSTRING

Armin Panjehpour
arminpp1379@gmail.com

Supervisor(s): Dr. Hajipour
Sharif University, Tehran, Iran

13/05/2022

Part.1 - Clustering using MLP & RBF

Here we use Multi Layer Perceptron and RBF networks to do clustering on a data.

Part.1.a - Data Visualization

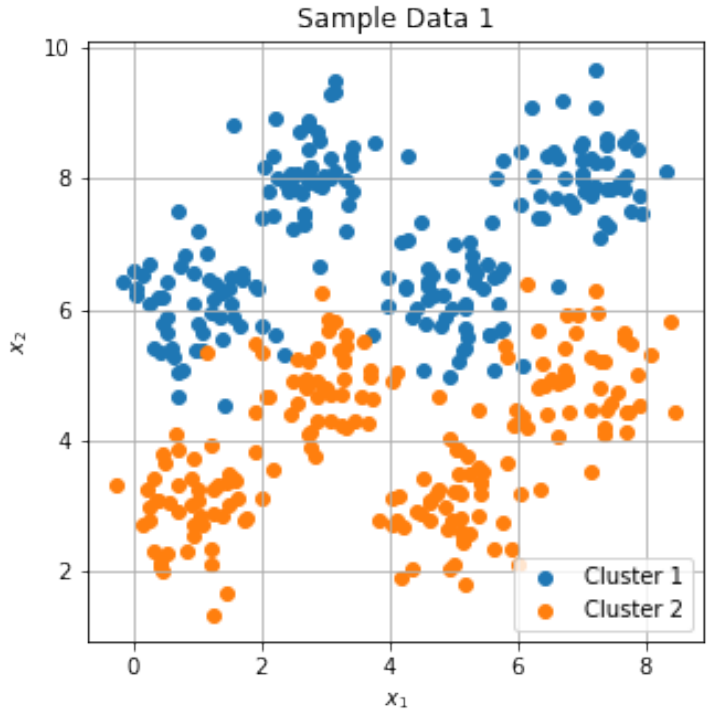

Figure 1: Sample Data 1

Part.1.b - Train Validation Data Split

As the HW says, we select 30 percentage of the data, randomly, as the validation data and the 70 percentage rest as the train data. We do this using sklearn model selection library. Here's the result:

```
(400, 2) (400, 1)
(280, 2) (120, 2) (280, 1) (120, 1)
```

As you can see, our data which has 400 samples is splitted in to two parts with 280 and 120 samples.

Part.1.c - Clustering using MLP

Here, we design a three layer perceptron two do clustering on train data. In order to design the MLP, we follow these steps:

Part.1.c.1 - Data Normalization

At first, we normalize all our data (both train and validation data) using train data mean and variance. For that, we use "StandardScaler from sklearn.preprocessing":

```
# Part.1.3 ------- MLP on data
# Normalize Data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(data_tr)
print(scaler.mean_.shape)
data_tr_norm = scaler.transform(data_tr)
data_val_norm = scaler.transform(data_val)
```

Part.1.c.2 - Build up the Network

Then we create our network. We use Sigmoid activation function for all neurons in hidden and output layer. Input layer will have two inputs for the sample's coordinates, Hidden layer will have 3 neurons and we'll have one output neuron.

```
# Part.1.3 ------- MLP on data
# CREATE MLP MODEL
# Setting up the layers
from tensorflow.keras.layers import Input, Dense, Activation

model = keras.Sequential([
                    Input(shape = (2,)), # input layer
                    Dense(units = 3), # hidden layer one
                    Activation(activation = tf.math.sigmoid), # fact = sigmoid
                    Dense(units = 1), # output layer
                    Activation(activation = tf.math.sigmoid), # fact = sigmoid
])

model.summary()
```

Part.1.c.3 - Compile the Network

Now we have to compile our network. We will use SGD(Stochastic Gradient Descent) with learning rate of 0.01 as our optimizer, MSE(Mean Squared Error) as the loss function and accuracy as our metric:

```
# Part.1.3 ------- MLP on data
# Compling the model

# make our model ready for training
# 1. optimizer 2. loss function 3. metrics

model.compile(
    optimizer = keras.optimizers.SGD(learning_rate = 0.01),
    loss = keras.losses.MeanSquaredError(),
    metrics = ['accuracy']
)
```

Part.1.c.4 - Train the Network

Now we start training the network with the training data in 50 epochs and measure the accuracy on validation data. We achieve the accuracy of 92 percentage which is very good on the train and validation data.

```
# Part.1.3 ------- MLP on data

from tensorflow.keras.callbacks import EarlyStopping

er_stop = EarlyStopping(monitor = 'val_loss', patience = 5, restore_best_weights = True)

# Training the model
# using .fit in keras
# inputs of .fit : 1. Train/Val Data 2. Batch Size(go over samples window size)
# 3. Number of Epochs(number of repeatation on all samples) 4. Callbacks
```

```
hist = model.fit(
    data_tr_norm,
    labels_tr,
    batch_size = 1,
    epochs = 50,
    validation_data = (data_val_norm, labels_val),
    callbacks = [er_stop]
)
```

Part.1.c.5 - MLP result

Here's the result of our network on train and validation data:

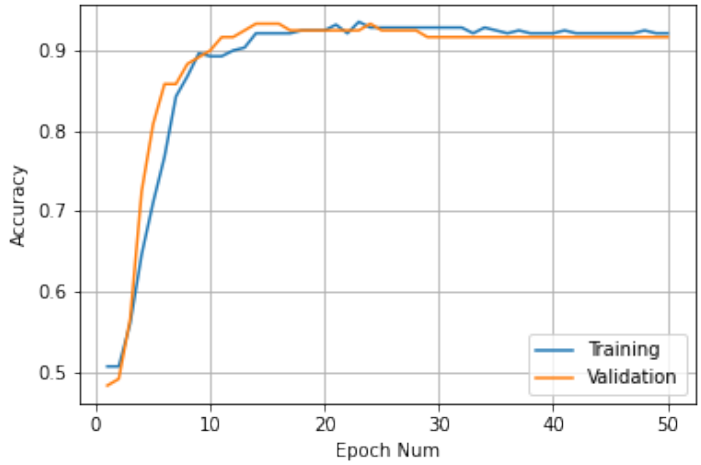

Figure 2: MLP Result

Part.1.d - Clustering using RBF

Here, we design a RBF network with two input neurons, one output neuron and a hidden layer consisting of 8 neurons. In order to design the RBF network, we've used the library TA suggested, (https://github.com/PetraVidnerova/rbf$_k eras$).

Initial clusters centers were created using "InitCentersKMeans" and not completely randomly as we saw both in lectures.

I've used MSE as the loss function and RMSprop(Root Mean Squared Propagation) as the network optimizer. We use the normalized training data to train the network:

```
# Part.1.4 ------- RBF on data
import keras.optimizers
from keras.models import Sequential
from keras.layers.core import Dense
from tensorflow.keras.optimizers import RMSprop
from rbflayer import RBFLayer, InitCentersRandom
from kmeans_initializer import InitCentersKMeans

model = Sequential()
rbfLayer = RBFLayer(8,
                    initializer = InitCentersKMeans(data_tr_norm),
                    betas = 1,
                    input_shape = (2,))

model.add(rbfLayer)
model.add(Dense(1))
```

```
model.compile(loss = 'mean_squared_error',
          optimizer = RMSprop())

model.fit(
data_tr_norm,
labels_tr,
batch_size = 1,
epochs = 50,
verbose=1)
```

We achieve the loss of 0.0352 on training data which is a very low loss which is less than the loss of the MLP as we could expected.

Part.1.d.1 - RBF Network Result on Validation Data

Here's the result of the network on validation data with a accuracy of 98.3 percentage:

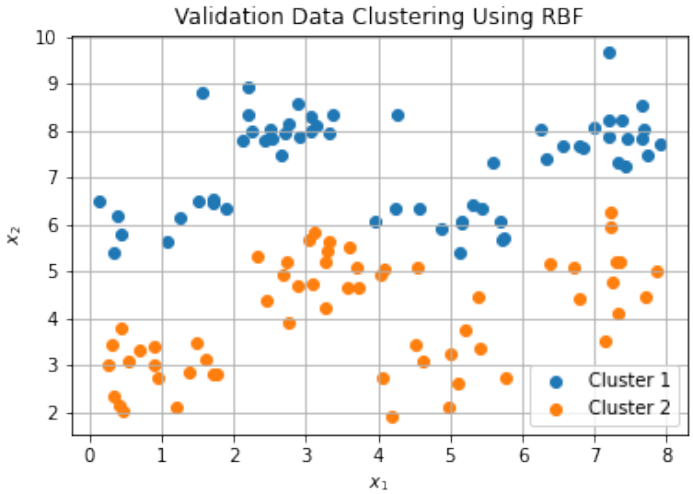

Figure 3: Validation Data Clustered Using RBF

```
# accuracy on validation data
y_pred = model.predict(data_val_norm)

y_pred[np.where(y_pred < 0.5)[0]] = 0
y_pred[np.where(y_pred >= 0.5)[0]] = 1

val_accuracy = np.count_nonzero((labels_val - y_pred) == 0)/ y_pred.shape[0]
print(val_accuracy)
    98.3
```

Part.1.e - Discussion

Since the data is not completely linearly separable, MLP won't give us 100 percentage accuracy but a high accuracy of 92 percentage. As you can see the data, it seems it will be seperated in to clusters using RBF too well. So applying a RBF network on our data leads us to the accuracy of 98.3 percentage which is higher than MLP. So, RBF is a better choose for clustering this data.