Computational Intelligence
HW2

Alireza Moradi

May 3, 2020

1 Train an neural network to learn $y = x^2$

In this question we had to train two neural networks to approximate $y = x^2$ function, one using MLP and another using RBF.

1.1 Use an MLP

1.1.1 Packages

Below are the packages used to implement this NN.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout
import keras
```

1.1.2 Data

Input data is 11000 random numbers between -100 and 100.

[ ]:
```
train_data_count = 11000
test_data_count = 300

x_train = np.random.uniform(low=-100,high=100,size=train_data_count)
x_train = np.sort(x_train)
y_train = np.power(x_train, 2)

x_train = np.reshape(x_train, (train_data_count, 1))
y_train = np.reshape(y_train, (train_data_count, 1))
```

1.1.3 Network Structure

A sequential model is used here with 2 hidden layers each with 120 and 40 neurons respectively. Both hidden layers are using *ReLU* as activation function. The output layer is a single neuron with linear output.

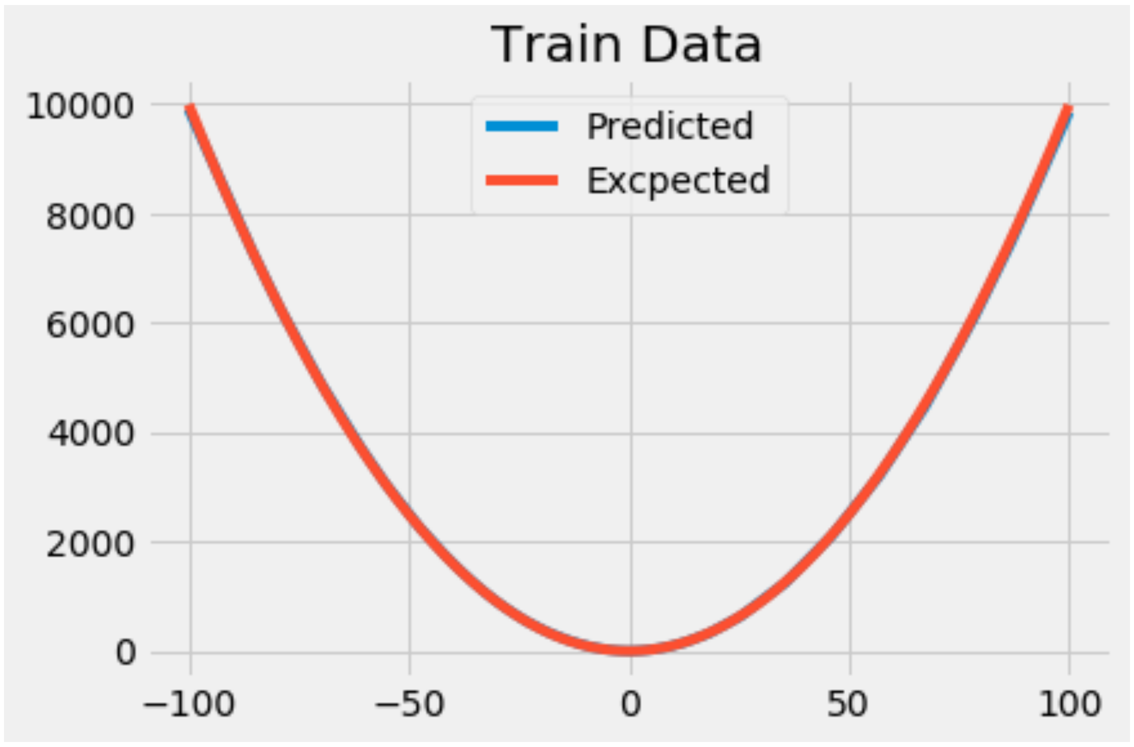
The model is then compiled with *Nadam* optimizer and *mean squared logarithmic error* is both its metric and loss function. Its then trained in 60 epochs and inputs are given to the network in batches of 15 samples.

[35]:
```
model = Sequential()
model.add(Dense(120,kernel_initializer='normal', activation='relu'))
model.add(Dense(40,kernel_initializer='normal', activation='relu'))
model.add(Dense(1,kernel_initializer='normal', activation='linear'))
model.compile(optimizer='Nadam', loss='msle', metrics=['msle'])
model.fit(x_train, y_train, batch_size=15, epochs=70)
```

```
Epoch 61/70
11000/11000 [==============================] - 1s 57us/step - loss: 1.4016e-04 -
msle: 1.4016e-04
Epoch 62/70
11000/11000 [==============================] - 1s 57us/step - loss: 1.5229e-04 -
msle: 1.5229e-04
Epoch 63/70
11000/11000 [==============================] - 1s 59us/step - loss: 1.1924e-04 -
msle: 1.1924e-04
Epoch 64/70
11000/11000 [==============================] - 1s 57us/step - loss: 1.0825e-04 -
msle: 1.0825e-04
Epoch 65/70
11000/11000 [==============================] - 1s 58us/step - loss: 1.1279e-04 -
msle: 1.1279e-04
Epoch 66/70
11000/11000 [==============================] - 1s 58us/step - loss: 1.1489e-04 -
msle: 1.1489e-04
Epoch 67/70
11000/11000 [==============================] - 1s 58us/step - loss: 1.2088e-04 -
msle: 1.2088e-04
Epoch 68/70
11000/11000 [==============================] - 1s 58us/step - loss: 9.9954e-05 -
msle: 9.9954e-05
Epoch 69/70
11000/11000 [==============================] - 1s 58us/step - loss: 2.1139e-04 -
msle: 2.1139e-04
Epoch 70/70
11000/11000 [==============================] - 1s 58us/step - loss: 6.3149e-05 -
msle: 6.3149e-05
```

Below the network approximation of the function vs. the actual function are plotted.

[52]:
```
output = model.predict(x_train)
plt.title("Train Data")
plt.plot(x_train,output,label='Predicted')
plt.plot(x_train,y_train,label='Excpected')
plt.legend()
plt.show()
```

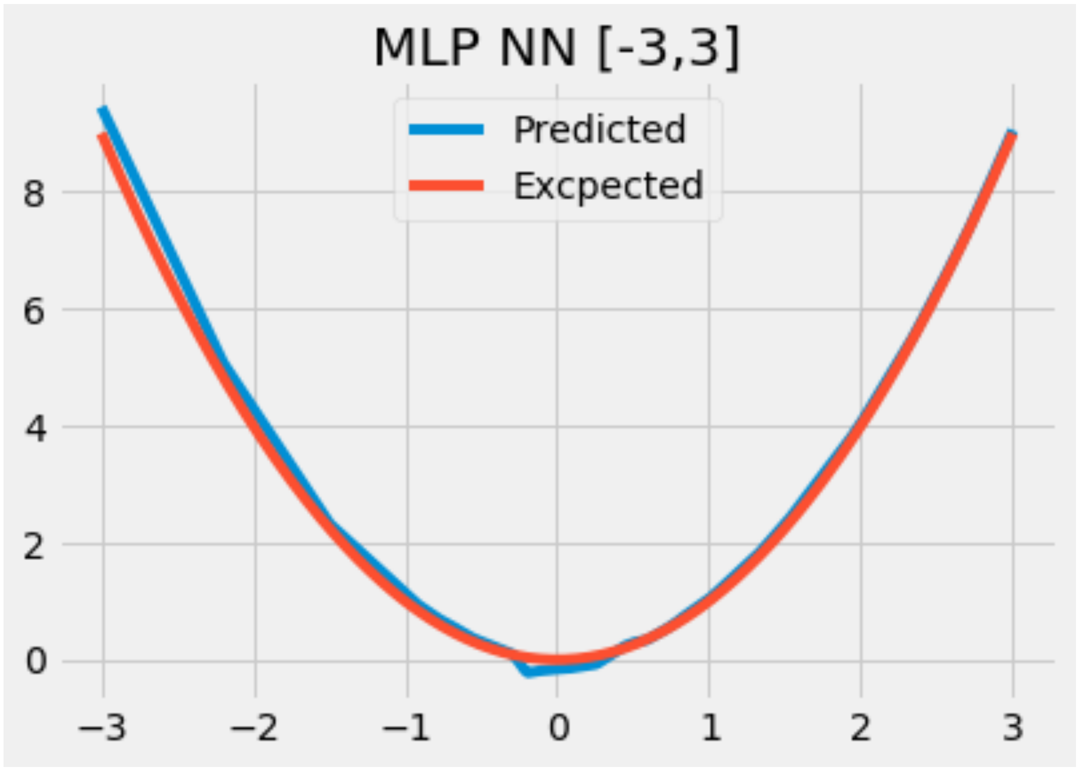


1.1.4 Test

Below is the approximation of the network vs. the actual function output in a range of [-3,3].

[54]:
```
test = np.linspace(-3,3, test_data_count)
test = np.reshape(test, (test_data_count, 1))
output = model.predict(test)
```

[55]:
```
plt.title("MLP NN [-3,3]")
plt.plot(test,output,label='Predicted')
plt.plot(test,np.power(test,2),label='Excpected')
plt.legend()
plt.show()
```



1.2 Use an RBF

1.2.1 Packages

Below are the packages used to implement an RBF network.

[39]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.2.2 Network Structure

RBFLayer is the class which represents the only hidden layer in the network which uses a Radial Function to map the input to a radial space.

- **bases** is the number of clusters as well as neurons of the hidden layer.
- **centers** is a vector of centers each cluster calculated in the *kmeans* function.
- ***stds*** is a vector of standard deviations of each cluster. This value is fixed for all of the clusters.
- ***A*** is a vector of outputs of neurons.

The input data is clustered here using k_means clustering algorithm and the center of each cluster is returned to use in the radial function.

```
[40]: class RBFLayer:
          def __init__(self, bases):
              self.bases = bases
              self.centers = np.ndarray((bases, 1))
              self.stds = np.ndarray((bases, 1))
              self.A = np.ndarray((bases, 1))

          def kmeans(self, X, k):
              clusters = np.random.choice(np.squeeze(X), size=k)
              prevClusters = clusters.copy()
              converged = False

              while not converged:
                  distances = np.squeeze(np.abs(X[:, np.newaxis] - clusters[np.
       ↪newaxis, :]))

                  closestCluster = np.argmin(distances, axis=1)

                  for i in range(k):
                      pointsForCluster = X[closestCluster == i]
                      if len(pointsForCluster) > 0:
                          clusters[i] = np.mean(pointsForCluster, axis=0)

                  converged = np.linalg.norm(clusters - prevClusters) < 1e-6
                  prevClusters = clusters.copy()

              distances = np.squeeze(np.abs(X[:, np.newaxis] - clusters[np.newaxis, :
       ↪]))
              closestCluster = np.argmin(distances, axis=1)

              clustersWithNoPoints = []
              for i in range(k):
                  pointsForCluster = X[closestCluster == i]
                  if len(pointsForCluster) < 2:
                      clustersWithNoPoints.append(i)
                      continue
              if len(clustersWithNoPoints) > 0:
                  pointsToAverage = []
                  for i in range(k):
                      if i not in clustersWithNoPoints:
                          pointsToAverage.append(X[closestCluster == i])

              return np.reshape(clusters, (self.bases, 1))
```

OutputLayer is the class which represents the single output neuron.

- $\boldsymbol{W}$ is weights corresponding to RBF layer's neurons output.

- ***A*** is the neuron's output.

```
[41]: class OutputLayer:
          def __init__(self, bases):
              self.W = np.zeros((bases, 1))
              self.A = 0
```

RBF is where the network is put together.

- ***bases*** is the number of radial functions which is actually number of RBF layers neurons and clusters.
- ***gaussian*** is the radial function used in the hidden layer calculated as $e^{(-std*(x_i - center_j)^2)}$
- ***linear*** is the linear activation function which multiplies hidden layers outputs by the weight and sums it all together.
- ***forward*** is where each layers output is computed.
- ***fit*** there is no training here, the inputs are given to the hidden layer and weights vector of the output layer is updated as $W = \sigma^{-1}.Y$ where σ is hidden layers output and Y is the actual output.

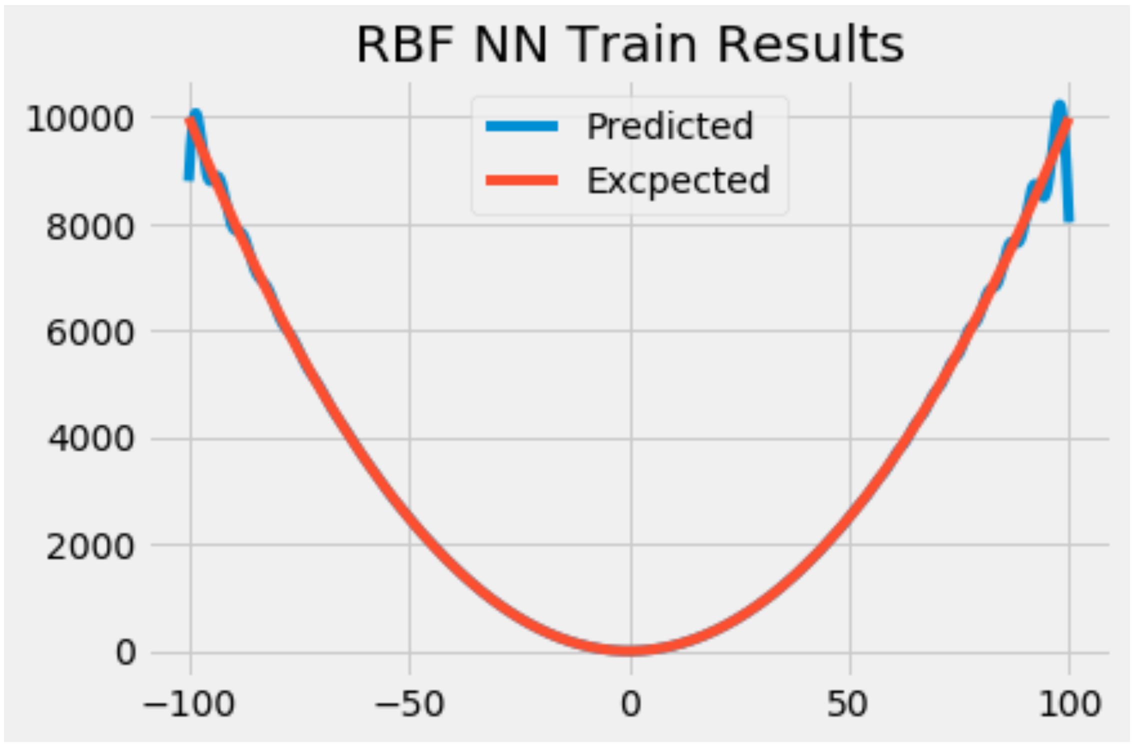
```
[42]: class RBF:
          def __init__(self, bases, sigma):
              self.bases = bases
              self.sigma = sigma
              self.hidden_layer = RBFLayer(bases)
              self.output_layer = OutputLayer(bases)

          def gaussian(self, x, c):
              return np.exp((-self.sigma * (x - c.T) ** 2))

          def linear(self):
              return self.hidden_layer.A.dot(self.output_layer.W)

          def forward(self, x):
              self.hidden_layer.A = self.gaussian(x, self.hidden_layer.centers)
              self.output_layer.A = self.linear()[0][0]

          def fit(self, input, output):
              self.hidden_layer.centers = self.hidden_layer.kmeans(input, self.bases)
              self.forward(input)
              self.hidden_layer.A = np.linalg.pinv(self.hidden_layer.A)
              self.output_layer.W = np.dot(self.hidden_layer.A, output)

          def predict(self, input):
              result = []
              for i in range(len(input)):
                  self.forward(input[i])
```

```
            result.append(self.output_layer.A)
        return result
```

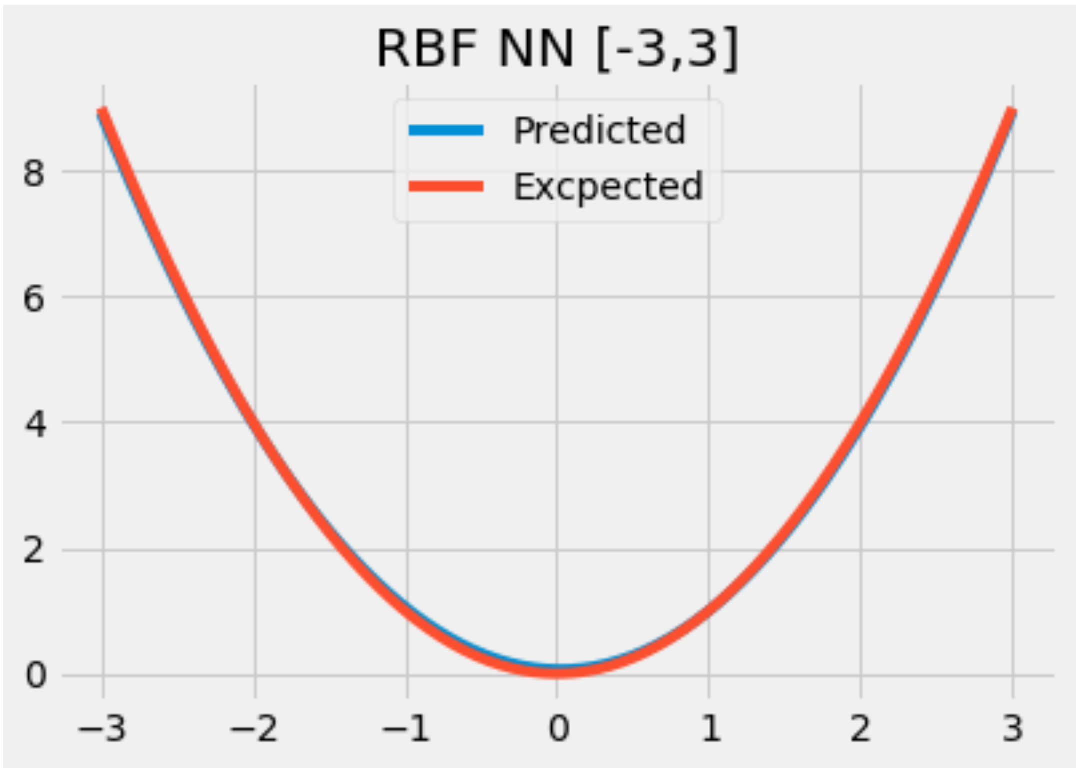
1.2.3 *main*

Input data is 11000 random numbers between −100 and 100 and the test data has 300 samples in range of [-3.3]. At last the approximation of the function and the actual output are plotted.

[45]:
```
if __name__ == '__main__':
    train_data_count = 11000
    test_data_count = 300

    x_train = np.random.uniform(low=-100, high=100, size=train_data_count)
    x_train = np.sort(x_train, axis=0)
    noise = np.random.uniform(-10, 10, train_data_count)
    y_train = np.power(x_train, 2)
    x_train = np.reshape(x_train, (train_data_count, 1))
    y_train = np.reshape(y_train, (train_data_count, 1))

    nn = RBF(bases=75, sigma=0.07)
    nn.fit(x_train, y_train)

    output = nn.predict(x_train)

    plt.title("RBF NN Train Results")
    plt.plot(x_train, output, label='Predicted')
    plt.plot(x_train, np.power(x_train, 2), label='Excpected')
    plt.legend()
    plt.show()
```



[46]:
```
test = np.linspace(-3, 3, test_data_count)
test = np.reshape(test, (test_data_count, 1))
output = nn.predict(test)

plt.title("RBF NN [-3,3]")
plt.plot(test, output, label='Predicted')
plt.plot(test, np.power(test, 2), label='Excpected')
plt.legend()
plt.show()
```



1.3 Compare the results of RBF and MLP

Because the RBF network doesn't actually need a training and weight vector is computed in just 1 iteration with very very good approximation it is significantly faster than MLP and the approximation of RBF is also better than mlp because it is proved that every function can be written as sum of several radial functions, So RBF network can have a very precise apprixmation of every function with the right number of radial functions.