Tutorial 02

SCS3201 / IS3117 / CS3120 Machine Learning and Neural Computing

Artificial Neural Networks

In this tutorial, we will be discussing what artificial neural network is and how it works. For that we will be using a dataset named **Breast Cancer Wisconsin Dataset.**

Read about the dataset from here.

This dataset has clinical data of 699 patients suspected of breast cancer. Data has been represented in 10 attributes which are,

1. Sample code number: id number
2. Clump Thickness: 1 - 10
3. Uniformity of Cell Size: 1 - 10
4. Uniformity of Cell Shape: 1 - 10
5. Marginal Adhesion: 1 - 10
6. Single Epithelial Cell Size: 1 - 10
7. Bare Nuclei: 1 - 10
8. Bland Chromatin: 1 - 10
9. Normal Nucleoli: 1 - 10
10. Mitoses: 1 - 10

And each instance is labeled into two classes **benign (2)** and **malignant (4).** This dataset will be used to try out the artificial neutral networks in this tutorial.

We will be using python to code the model from scratch.

Download the dataset from here.

We are using only numpy for constructing this Artificial Neural Network. Let's import numpy and load the dataset.

```
# import numpy
import numpy as np

# load the dataset
data = np.genfromtxt('breast-cancer-wisconsin.data', delimiter=',')
```

If you have read the description of the dataset, you should've noticed that this dataset contains missing values. There are many ways to deal with missing values, which you can find here.

For this example, we are deleting the rows which have missing values.

```
# remove all the rows with nan value
data = data[~np.isnan(data).any(axis=1)]
```

The 1st column of the dataset has the sample code number which is not an attribute needed for predicting breast cancer. So let's remove that column as well.

```
# remove the id column
data = np.delete(data, 0, axis=1)
```

Now we get a dataset with no missing values, 9 attributes and 1 class label. The label is 2 for **benign** and 4 for **malignant.** Since this is a binary case, we change those labels to 0 and 1.

```
# replace class with 0 and 1
data[:, 9][data[:, 9] == 2] = 0
data[:, 9][data[:, 9] == 4] = 1
```

Now our dataset has binary labels as well. Now we need to normalize the dataset. We are only normalizing the attributes. Hence, we shuffle the rows and divide the dataset in to two parts named attributes and labels.

```
#shuffle data and divide to attributes and labels
np.random.shuffle(data)
attributes, labels = data[:, :9], data[:, 9:]
```

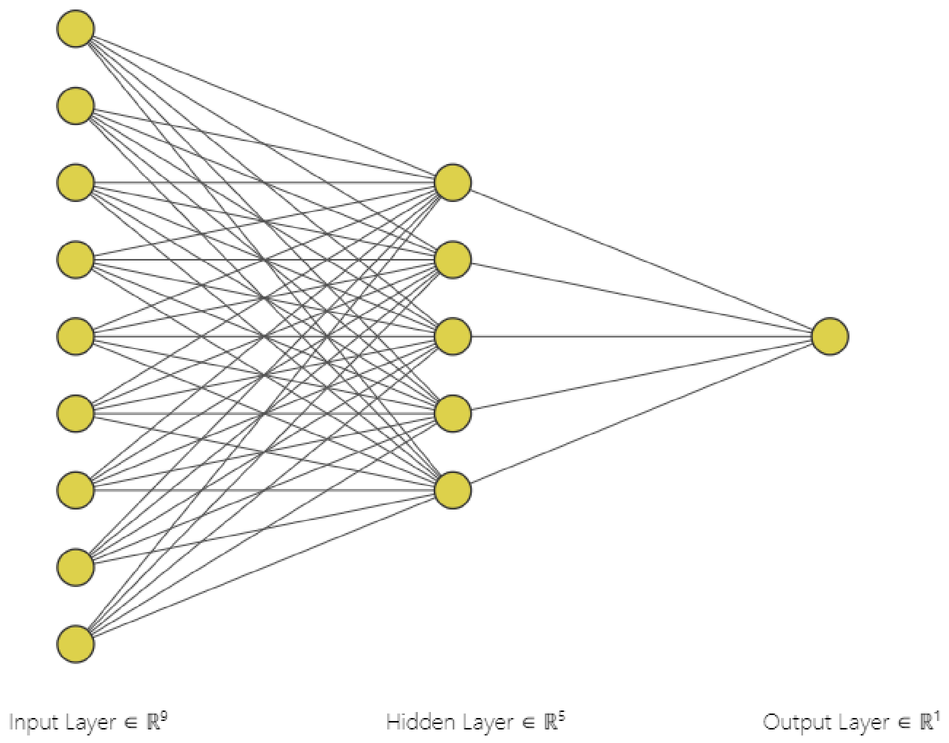
Now we are normalizing the attributes. For that there are several normalization methods which you can find here. Here we are using MinMax method to normalize the attributes.

```
#normalize the attributes
x_max, x_min = attributes.max(), attributes.min()
attributes = (attributes - x_min)/(x_max - x_min)
```

As the final step of the data preprocessing, we divide the dataset in to training set (70%) and testing set (30%).

```
# divide dataset into training 70% and testing 30%
margin = len(data)//10*7
training_x, testing_x = attributes[:margin, :], attributes[margin:, :]
training_y, testing_y = labels[:margin, :], labels[margin:, :]
```

Now our dataset is ready. Let’s dive into the Neural Network. Here we have 9 attributes, therefore input layer would have 9 neurons. Output is a binary value, hence output layer would have only one neuron. And for this tutorial we’ll have one hidden layer. Let’s put 5 neurons in the hidden layer.



In this example we will be constructing this neural network without biases for simplicity. Initially we initialize the weights randomly.

```
class NeuralNetwork:

    def __init__(self):
        # initialize weights with random values
        self.weights1 = np.random.rand(9, 5)
        self.weights2 = np.random.rand(5, 1)

        # declare variables for pred_output, input and labels
        self.output = None
        self.input = None
        self.y = None
```

Then we need an activation function. We are using **Sigmoid Function** as the activation function. Therefore, we need to implement both sigmoid function and its derivative.

$$S(x) = \frac{1}{1 + e^{-x}} \qquad \acute{S}(x) = S(x)\,(1 - S(x))$$

```
def sigmoid(self, Z):
        return 1/(1+np.exp(-Z))

def sigmoid_derivative(self, Z):
        return Z * (1 - Z)
```

Then we can define the feedforward function which uses above initialized weights and activation function.

```
    def feedforward(self):
        # feedforward to layer 1
        self.layer1 = self.sigmoid(np.dot(self.input, self.weights1))

        # feedforward to layer 2
        self.layer2 = self.sigmoid(np.dot(self.layer1, self.weights2))

        return self.layer2
```

Now we can give some input and get the output for randomly initialize weights. But we are creating a neural network which can learn. So it should have the ability to change it's weights minimizing the error.

That is achieved by the back propagation. We are using gradient decent just like we used in linear regression. For each weight we get partial derivative for the cost function and update weights with respect to the learning rate.

```
    def backprop(self, learning_rate):
        # get partial derivative for layer 2 weights
        d_weights2 = np.dot(self.layer1.T, 2*(self.y - self.output)
                        * self.sigmoid_derivative(self.output))

        # get partial derivative for layer 1 weights
        d_weights1 = np.dot(self.input.T, np.dot(2*(self.y - self.output)
                        *self.sigmoid_derivative(self.output), self.weights2.T)
                        *self.sigmoid_derivative(self.layer1))

        # adjust weights
        self.weights1 += learning_rate*d_weights1
        self.weights2 += learning_rate*d_weights2
```

Now we can train the network and check how it performs for test data.

```
    def train(self, X, y, learning_rate):
        # set input and labels
        self.input = X
        self.y = y

        #feedforward and setoutput
        self.output = self.feedforward()

        # backpropagate
        self.backprop(learning_rate)

        # return training error
        return np.mean(np.square(y - np.round(self.output)))

    def test(self, X, y):
        # set input and labels
        self.input = X
        self.y = y

        #feedforward and setoutput
        self.output = self.feedforward()

        # print test results
        print("\nTesting Results\nError : " +
              str(np.mean(np.square(y - np.round(self.output))))+"\n")
```

That is all for the neural network implementation. Now we need data to make this network learn and we have preprocessed data which can be use for it. We'll use the training set to train the dataset and we'll use 0.01 as the learning rate.

Let's train this network for 1000 iterations and check it's accuracy.

```
NN = NeuralNetwork()
learning_rate = 0.01

# train this network for 1000 iterations
for i in range(1, 1001):
    error = NN.train(training_x, training_y, learning_rate)

    # print error after every 10 iterations
    if i % 10 == 0 or i == 1:
        print("Iteration : "+str(i)+" | Error : "+str(error))

NN.test(testing_x, testing_y)
```

After training the network for 1000 iterations, let's get the testing set and check the accuracy of the neural networks' prediction.

The output should be something like this. Error is minimized with each iteration.

```
Iteration : 1 | Error : 0.6239495798319328
Iteration : 10 | Error : 0.3760504201680672
Iteration : 20 | Error : 0.3760504201680672
Iteration : 30 | Error : 0.3760504201680672
……………………………………………………………………………………………………………………
……………………………………………………………………………………………………………………
Iteration : 950 | Error : 0.021008403361344S4
Iteration : 960 | Error : 0.02100840336134454
Iteration : 970 | Error : 0.02100840336134454
Iteration : 980 | Error : 0.02100840336134454
Iteration : 990 | Error : 0.02100840336134454
Iteration : 1000 | Error : 0.02100840336134454

Testing Results
Error : 0.01932367149758454
```

The error values could change with each execution of the program since we have used random weights to initialize the network.

Try changing the number of neurons in the hidden layer, learning rate, activation function and method of initializing weights to see how results change with those parameters.

You can find the full implementation here.

If you have any clarifications to make, please post them in the discussion forum.

****