Backward propagation:

1) Script:import numpy,we define our neural network as a class,with constructor taking in layers(a list of nodes in each layer) and alpha(optional) parameters.The weight initialized to an empty list while the layers to the number of layers provided and alpha having a default value of 0.1

```
# import the necessary packages
import numpy as np

class NeuralNetwork:
    def __init__(self, layers, alpha=0.1):
        # initialize the list of weights matrices, then store the
        # network architecture and learning rate
        self.W = []
        self.layers = layers
        self.alpha = alpha
```

2) Script:We get the dimension of the weight matrix from the (N-2) layers and then populate the weight matrix by dividing the randomly generated value with the square root of the number of nodes in each layer so as to normalize the variance output

```
        # start looping from the index of the first layer but
        # stop before we reach the last two layers
        for i in np.arange(0, len(layers) - 2):
            # randomly initialize a weight matrix connecting the
            # number of nodes in each respective layer together,
            # adding an extra node for the bias
            w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
            self.W.append(w / np.sqrt(layers[i]))
```

3) Script: for the last two layers which help to handle the bias part we also randomly generated values and normalize the value before storing it to the weight matrix

```
        # the last two layers are a special case where the input
        # connections need a bias term but the output does not
        w = np.random.randn(layers[-2] + 1, layers[-1])
        self.W.append(w / np.sqrt(layers[-2]))
```

4) Script:we define the magic function for this neural network class and all it does is to print the number of nodes in each layer as string.

```
    def __repr__(self):
        # construct and return a string that represents the network
        # architecture
        return "NeuralNetwork: {}".format(
            "-".join(str(l) for l in self.layers))
```

5) Script:we define our activation function(sigmoid)

```
    def sigmoid(self, x):
        # compute and return the sigmoid activation value for a
        # given input value
        return 1.0 / (1 + np.exp(-x))
```

6) Script:the define the derivative of sigmoid function

```
    def sigmoid_deriv(self, x):
        # compute the derivative of the sigmoid function ASSUMING
        # that `x` has already been passed through the `sigmoid`
        # function
        return x * (1 - x)
```

7) Script: we define fit function which takes 4 parameters,where two out of the four are optional.We make the bias trainable with the weight by adding it to the input parameter For each epoch,we predict the input,calculate the backward propagation phase and then update the weight matrix.The last few line check if we should display the loss or not

```
    def fit(self, X, y, epochs=1000, displayUpdate=100):
        # insert a column of 1's as the last entry in the feature
        # matrix -- this little trick allows us to treat the bias
        # as a trainable parameter within the weight matrix
        X = np.c_[X, np.ones((X.shape[0]))]

        # loop over the desired number of epochs
        for epoch in np.arange(0, epochs):
            # loop over each individual data point and train
            # our network on it
            for (x, target) in zip(X, y):
                self.fit_partial(x, target)

            # check to see if we should display a training update
            if epoch == 0 or (epoch + 1) % displayUpdate == 0:
                loss = self.calculate_loss(X, y)
                print("[INFO] epoch={}, loss={:.7f}".format(
                    epoch + 1, loss))
```

8) Script:we define fit_partial method where the backpropagation takes place.It takes two arguments: the input data and corresponding target.A list **A** is defined to store the output of activation for each layer. Here we initialize it as the input data which is actually the first activation input.

```
   # construct our list of output activations for each layer
        # as our data point flows through the network; the first
        # activation is a special case -- it's just the input
        # feature vector itself
        A = [np.atleast_2d(x)]
```

9) Script: we loop through each layer,find the net input by computing the dot product of it activation with it weight.The output of the layer is the sigmoid of the net input .The output is then appended to the activation list.The process is repeated for other layers,the final out of activation is the prediction.The whole process account for the feed forward of the learning.

```
# FEEDFORWARD:
        # loop over the layers in the network
        for layer in np.arange(0, len(self.W)):
            # feedforward the activation at the current layer by
            # taking the dot product between the activation and
            # the weight matrix -- this is called the "net input"
            # to the current layer
            net = A[layer].dot(self.W[layer])

            # computing the "net output" is simply applying our
            # non-linear activation function to the net input
            out = self.sigmoid(net)

            # once we have the net output, add it to our list of
            # activations
            A.append(out)
```

10) Script:After getting the output for the epoch,we dive next to the backwardpropagation. Since the last activation is our prediction we find the error as below and apply chain rule

to build delta which will be used to update the weight matrix.So we start with the delta of the last layer and then move backward.

```
        # BACKPROPAGATION
        # the first phase of backpropagation is to compute the
        # difference between our *prediction* (the final output
        # activation in the activations list) and the true target
        # value
        error = A[-1] - y

        # from here, we need to apply the chain rule and build our
        # list of deltas `D`; the first entry in the deltas is
        # simply the error of the output layer times the derivative
        # of our activation function for the output value
        D = [error * self.sigmoid_deriv(A[-1])]
```

11) Script:we loop through the layers ignoring the previous two layer which has been accounted for in the last line above.Here we move backward,computing the delta for the current layer which equals the dot product of previous layer delta and the weight matrix of the current layer multiplied by the derivative of the activation output of the layer.The current delta is then appended to the list of delta.

```
for layer in np.arange(len(A) - 2, 0, -1):
            # the delta for the current layer is equal to the delta
            # of the *previous layer* dotted with the weight matrix
            # of the current layer, followed by multiplying the
delta
            # by the derivative of the non-linear activation
function
            # for the activations of the current layer
            delta = D[-1].dot(self.W[layer].T)
            delta = delta * self.sigmoid_deriv(A[layer])
            D.append(delta)
```

12) Script:we reverse the delta list since we got them backward.Then we loop through the delta to update our weight for each layer which is the gradient descent.

```
        # since we looped over our layers in reverse order we need
to
        # reverse the deltas
        D = D[::-1]

        # WEIGHT UPDATE PHASE
        # loop over the layers
        for layer in np.arange(0, len(self.W)):
            # update our weights by taking the dot product of the
layer
            # activations with their respective deltas, then
multiplying
            # this value by some small learning rate and adding to
our
            # weight matrix -- this is where the actual "learning"
takes
            # place
            self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

13) Next we define a predict method which takes one argument and returns the predicted value.It is just the similar process done in fit method, just that the out is returned instead of appending it to the activation list.

```
    def predict(self, X, addBias=True):
        # initialize the output prediction as the input features --
this
        # value will be (forward) propagated through the network to
        # obtain the final prediction
        p = np.atleast_2d(X)

        # check to see if the bias column should be added
        if addBias:
            # insert a column of 1's as the last entry in the
feature
            # matrix (bias)
            p = np.c_[p, np.ones((p.shape[0]))]

        # loop over our layers in the network
        for layer in np.arange(0, len(self.W)):
            # computing the output prediction is as simple as taking
            # the dot product between the current activation value
`p`
```

```
            # and the weight matrix associated with the current
layer,
            # then passing this value through a non-linear
activation
            # function
            p = self.sigmoid(np.dot(p, self.W[layer]))

        # return the predicted value
        return p
```

14) Script:Calculate_loss method takes in input data and corresponding target.we make prediction on the input data and calculate the loss against the target.then we return the loss.

```
    def calculate_loss(self, X, targets):
        # make predictions for the input data points then compute
        # the loss
        targets = np.atleast_2d(targets)
        predictions = self.predict(X, addBias=False)
        loss = 0.5 * np.sum((predictions - targets) ** 2)

        # return the loss
        return loss
```

Applying the neural network on nonlinear function-XOR dataset

1) Script:import our neural network class and setting up our XOR data

```
# import the necessary packages
from pyimagesearch.nn import NeuralNetwork
import numpy as np

# construct the XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

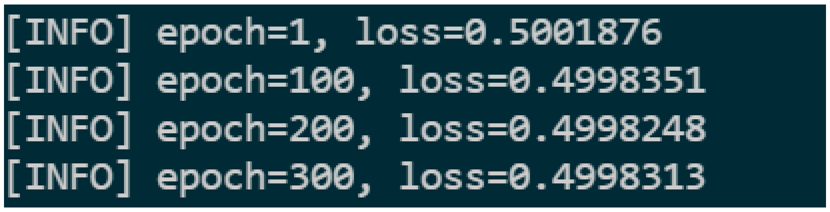
2) Script:we create and object from our Neural network class with the layer architecture and alpha set,then train the neural network using the data defined above, and specified epoch

```
# define our 2-2-1 neural network and train it
nn = NeuralNetwork([2, 2, 1], alpha=0.5)
nn.fit(X, y, epochs=20000)
```

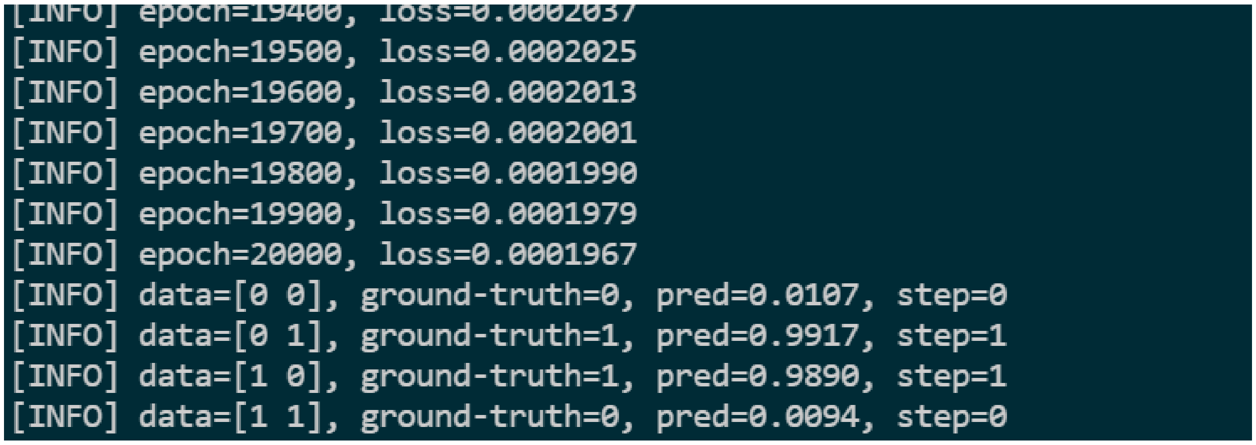
3) Script: we predict each entry in the dataset using the predict method of our neural network,then we apply step function and output the result

```
# now that our network is trained, loop over the XOR data points
for (x, target) in zip(X, y):
    # make a prediction on the data point and display the result
    # to our console
    pred = nn.predict(x)[0][0]
    step = 1 if pred > 0.5 else 0
    print("[INFO] data={}, ground-truth={}, pred={:.4f},
step={}".format(
        x, target[0], pred, step))
```

Result: From the result, our loss was greatly reduced by from 0.5 to 0.00019 and the predict on the data set is very accurate



```
[INFO] epoch=1, loss=0.5001876
[INFO] epoch=100, loss=0.4998351
[INFO] epoch=200, loss=0.4998248
[INFO] epoch=300, loss=0.4998313
```



```
[INFO] epoch=19400, loss=0.0002037
[INFO] epoch=19500, loss=0.0002025
[INFO] epoch=19600, loss=0.0002013
[INFO] epoch=19700, loss=0.0002001
[INFO] epoch=19800, loss=0.0001990
[INFO] epoch=19900, loss=0.0001979
[INFO] epoch=20000, loss=0.0001967
[INFO] data=[0 0], ground-truth=0, pred=0.0107, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9917, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9890, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0094, step=0
```

MNIST handwriting dataset:We apply the neural network to

1) Script: importing necessary package

```
# import the necessary packages
```

```
from pyimagesearch.nn import NeuralNetwork
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn import datasets
```

2) Script:we import the dataset,then we scale using min-max normalization so that each image have a pixel intensity between 0 and 1.

```
print("[INFO] loading MNIST (sample) dataset...")
digits = datasets.load_digits()
data = digits.data.astype("float")
data = (data - data.min()) / (data.max() - data.min())
print("[INFO] samples: {}, dim: {}".format(data.shape[0],
    data.shape[1]))
```

3) Script:we split our data into train and test data,and encoding the train and test label

```
# construct the training and testing splits
(trainX, testX, trainY, testY) = train_test_split(data,
    digits.target, test_size=0.25)

# convert the labels from integers to vectors
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

4) Script: We create an object for the neural network with the the layer architecture to be used.Then we train our data on the neural network created

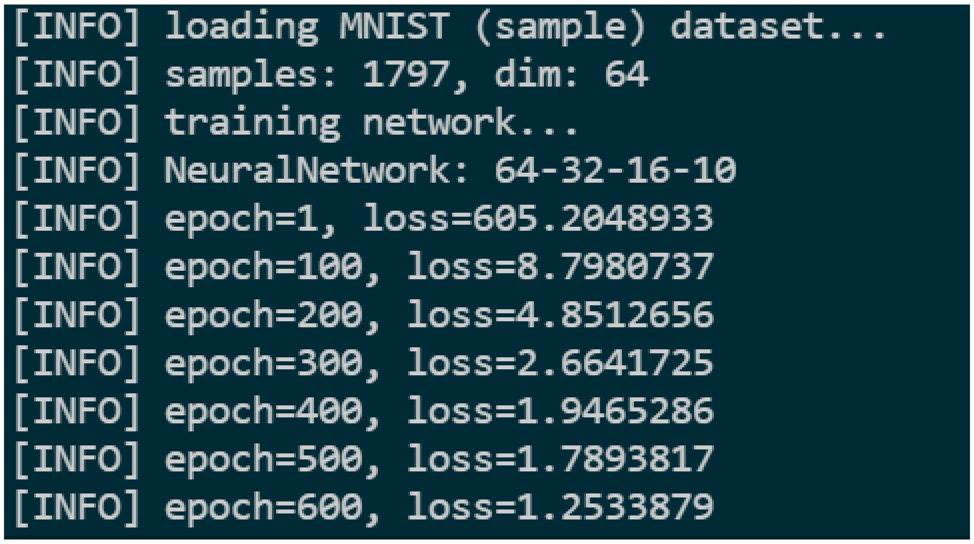
```
#train the network
print("[INFO] training network...")
nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])
print("[INFO] {}".format(nn))
nn.fit(trainX, trainY, epochs=1000)
```

5) Script:We make prediction on the test data,and obtain a class with the maximum probability.and print the report

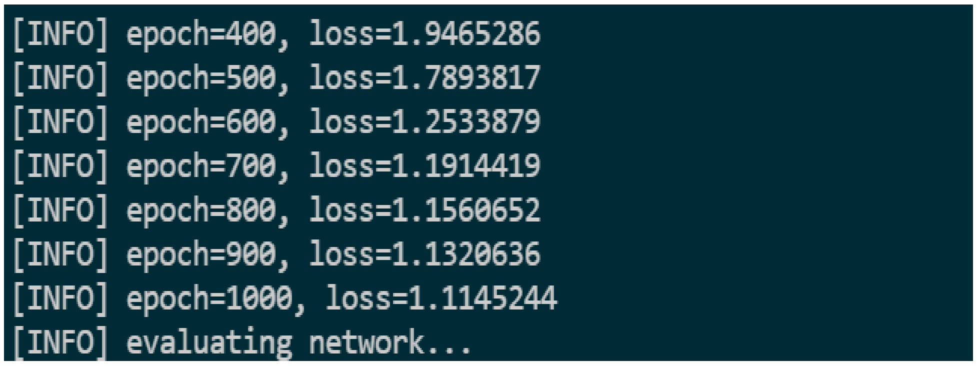
```
# evaluate the network
print("[INFO] evaluating network...")
```

```
predictions = nn.predict(testX)
predictions = predictions.argmax(axis=1)
print(classification_report(testY.argmax(axis=1),
predictions))
```

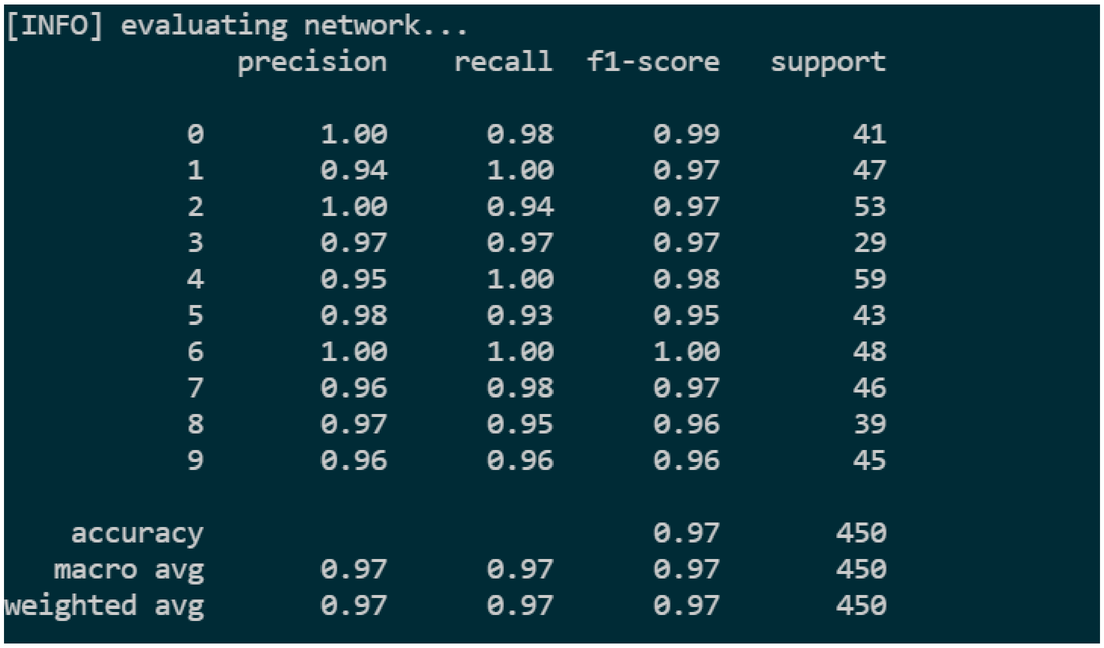
Result:the neural network really minimize the loss and gave an accuracy of 97%.



```
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-32-16-10
[INFO] epoch=1, loss=605.2048933
[INFO] epoch=100, loss=8.7980737
[INFO] epoch=200, loss=4.8512656
[INFO] epoch=300, loss=2.6641725
[INFO] epoch=400, loss=1.9465286
[INFO] epoch=500, loss=1.7893817
[INFO] epoch=600, loss=1.2533879
```



```
[INFO] epoch=400, loss=1.9465286
[INFO] epoch=500, loss=1.7893817
[INFO] epoch=600, loss=1.2533879
[INFO] epoch=700, loss=1.1914419
[INFO] epoch=800, loss=1.1560652
[INFO] epoch=900, loss=1.1320636
[INFO] epoch=1000, loss=1.1145244
[INFO] evaluating network...
```



```
[INFO] evaluating network...
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        41
           1       0.94      1.00      0.97        47
           2       1.00      0.94      0.97        53
           3       0.97      0.97      0.97        29
           4       0.95      1.00      0.98        59
           5       0.98      0.93      0.95        43
           6       1.00      1.00      1.00        48
           7       0.96      0.98      0.97        46
           8       0.97      0.95      0.96        39
           9       0.96      0.96      0.96        45

    accuracy                           0.97       450
   macro avg       0.97      0.97      0.97       450
weighted avg       0.97      0.97      0.97       450
```