EXPERIMENT- 7

AIM
Implement a back propagation problem on a given dataset.

SOFTWARE USED
Google Colab Platform - Python Programming Language

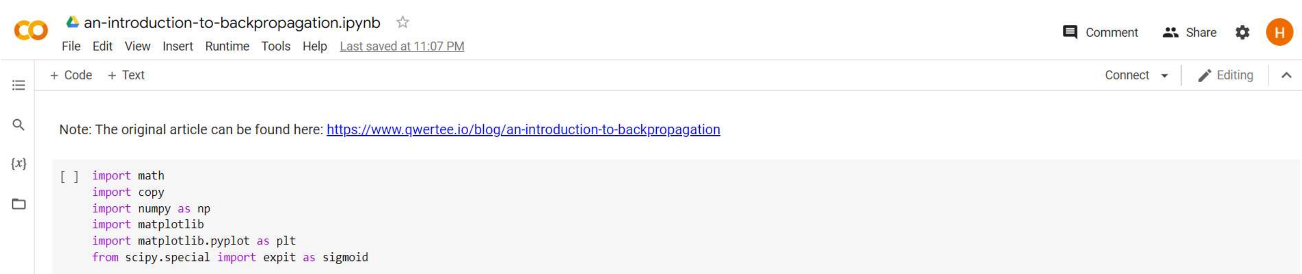
PROGRAM CODE

```
import math
import copy
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid

np.random.seed(0)

def generate_dataset(N_points):
    # 1 class
    radiuses = np.random.uniform(0, 0.5, size=N_points//2)
    angles = np.random.uniform(0, 2*math.pi, size=N_points//2)

    x_1 = np.multiply(radiuses, np.cos(angles)).reshape(N_points//2, 1)
    x_2 = np.multiply(radiuses, np.sin(angles)).reshape(N_points//2, 1)
    X_class_1 = np.concatenate((x_1, x_2), axis=1)
    Y_class_1 = np.full((N_points//2,), 1)

    # 0 class
    radiuses = np.random.uniform(0.6, 1, size=N_points//2)
    angles = np.random.uniform(0, 2*math.pi, size=N_points//2)

    x_1 = np.multiply(radiuses, np.cos(angles)).reshape(N_points//2, 1)
    x_2 = np.multiply(radiuses, np.sin(angles)).reshape(N_points//2, 1)
    X_class_0 = np.concatenate((x_1, x_2), axis=1)
    Y_class_0 = np.full((N_points//2,), 0)

    X = np.concatenate((X_class_1, X_class_0), axis=0)
    Y = np.concatenate((Y_class_1, Y_class_0), axis=0)
    return X, Y

N_points = 1000
X, Y = generate_dataset(N_points)

plt.scatter(X[:N_points//2, 0], X[:N_points//2, 1], color='red', label='class 1')
plt.scatter(X[N_points//2:, 0], X[N_points//2:, 1], color='blue', label='class 0')
plt.legend(loc=9, bbox_to_anchor=(0.5, -0.1), ncol=2)
plt.show()

weights = {
```

```
    'W1': np.random.randn(3, 2),
    'b1': np.zeros(3),
    'W2': np.random.randn(3),
    'b2': 0,
}

def forward_propagation(X, weights):
    # this implement the vectorized equations defined above.
    Z1 = np.dot(X, weights['W1'].T)  + weights['b1']
    H = sigmoid(Z1)
    Z2 = np.dot(H, weights['W2'].T) + weights['b2']
    Y = sigmoid(Z2)
    return Y, Z2, H, Z1

def back_propagation(X, Y_T, weights):
    N_points = X.shape[0]

    # forward propagation
    Y, Z2, H, Z1 = forward_propagation(X, weights)
    L = (1/N_points) * np.sum(-Y_T * np.log(Y) - (1 - Y_T) * np.log(1 - Y))
    # back propagation
    dLdY = 1/N_points * np.divide(Y - Y_T, np.multiply(Y, 1-Y))
    dLdZ2 = np.multiply(dLdY, (sigmoid(Z2)*(1-sigmoid(Z2))))
    dLdW2 = np.dot(H.T, dLdZ2)
    dLdb2 = np.dot(dLdZ2.T, np.ones(N_points))
    dLdH = np.dot(dLdZ2.reshape(N_points, 1), weights['W2'].reshape(1, 3))
    dLdZ1 = np.multiply(dLdH, np.multiply(sigmoid(Z1), (1-sigmoid(Z1))))
    dLdW1 = np.dot(dLdZ1.T, X)
    dLdb1 = np.dot(dLdZ1.T, np.ones(N_points))

    gradients = {
        'W1': dLdW1,
        'b1': dLdb1,
        'W2': dLdW2,
        'b2': dLdb2,
    }
    return gradients, L


epochs = 2000
epsilon = 1
initial_weights = copy.deepcopy(weights)

losses = []
for epoch in range(epochs):
    gradients, L = back_propagation(X, Y, weights)
    for weight_name in weights:
        weights[weight_name] -= epsilon * gradients[weight_name]

    losses.append(L)

plt.scatter(range(epochs), losses)
plt.title('Training Loss')
```

```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

def visualization(weights, X_data, title, superposed_training=False):
    N_test_points = 1000
    xs = np.linspace(1.1*np.min(X_data), 1.1*np.max(X_data), N_test_points)
    datapoints = np.transpose([np.tile(xs, len(xs)), np.repeat(xs, len(xs))])
    Y_initial = forward_propagation(datapoints, weights)[0].reshape(N_test_points, N_test_points)
    X1, X2 = np.meshgrid(xs, xs)
    plt.pcolormesh(X1, X2, Y_initial)
    plt.colorbar(label='P(1)')
    if superposed_training:
        plt.scatter(X_data[:N_points//2, 0], X_data[:N_points//2, 1], color='red')
        plt.scatter(X_data[N_points//2:, 0], X_data[N_points//2:, 1], color='blue')
    plt.title(title)
    plt.show()

visualization(initial_weights, X,  'Visualization before learning')
visualization(weights, X, 'Visualization after learning')
visualization(weights, X, 'Visualization after learning', superposed_training=True)
```

OUTPUT



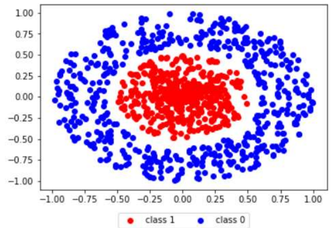
an-introduction-to-backpropagation.ipynb

File Edit View Insert Runtime Tools Help Last saved at 11:07 PM

Comment Share

+ Code + Text Connect Editing

```python
plt.scatter(X[:N_points//2, 0], X[:N_points//2, 1], color='red', label='class 1')
plt.scatter(X[N_points//2:, 0], X[N_points//2:, 1], color='blue', label='class 0')
plt.legend(loc=9, bbox_to_anchor=(0.5, -0.1), ncol=2)
plt.show()
```



an-introduction-to-backpropagation.ipynb

File Edit View Insert Runtime Tools Help Last saved at 11:07 PM

Comment Share

+ Code + Text Connect Editing

Forward propagation

Let's now implement the code for the forward propagation through the network.

```python
weights = {
    'W1': np.random.randn(3, 2),
    'b1': np.zeros(3),
    'W2': np.random.randn(3),
    'b2': 0,
}

def forward_propagation(X, weights):
    # this implement the vectorized equations defined above.
    Z1 = np.dot(X, weights['W1'].T)  + weights['b1']
    H = sigmoid(Z1)
    Z2 = np.dot(H, weights['W2'].T) + weights['b2']
    Y = sigmoid(Z2)
    return Y, Z2, H, Z1
```

an-introduction-to-backpropagation.ipynb

File Edit View Insert Runtime Tools Help Last saved at 11:07 PM

Comment Share

+ Code + Text Connect Editing

We can now define the code for the backpropagation:

```python
def back_propagation(X, Y_T, weights):
    N_points = X.shape[0]

    # forward propagation
    Y, Z2, H, Z1 = forward_propagation(X, weights)
    L = (1/N_points) * np.sum(-Y_T * np.log(Y) - (1 - Y_T) * np.log(1 - Y))
    # back propagation
    dLdY = 1/N_points * np.divide(Y - Y_T, np.multiply(Y, 1-Y))
    dLdZ2 = np.multiply(dLdY, (sigmoid(Z2)*(1-sigmoid(Z2))))
    dLdW2 = np.dot(H.T, dLdZ2)
    dLdb2 = np.dot(dLdZ2.T, np.ones(N_points))
    dLdH = np.dot(dLdZ2.reshape(N_points, 1), weights['W2'].reshape(1, 3))
    dLdZ1 = np.multiply(dLdH, np.multiply(sigmoid(Z1), (1-sigmoid(Z1))))
    dLdW1 = np.dot(dLdZ1.T, X)
    dLdb1 = np.dot(dLdZ1.T, np.ones(N_points))


    gradients = {
        'W1': dLdW1,
        'b1': dLdb1,
        'W2': dLdW2,
        'b2': dLdb2,
    }
    return gradients, L
```

Training: gradient descent

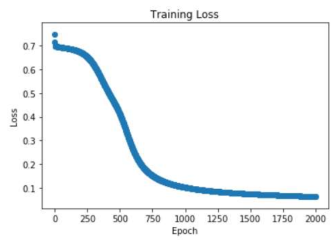
We have all in place to start training our network using gradient descent. Remember, at every iteration the weights and the biases are updated as $w^{(n+1)} = w^{(n)} - \epsilon \frac{\partial L}{\partial w}$.

```python
epochs = 2000
epsilon = 1
initial_weights = copy.deepcopy(weights)
```

an-introduction-to-backpropagation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

```python
initial_weights = copy.deepcopy(weights)

losses = []
for epoch in range(epochs):
    gradients, L = back_propagation(X, Y, weights)
    for weight_name in weights:
        weights[weight_name] -= epsilon * gradients[weight_name]

    losses.append(L)

plt.scatter(range(epochs), losses)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



As we can see in the plot above where the loss is plotted with respect to the number of epochs the network experienced, we clearly observed a decrease of the loss. In other words, the network seems to make less and less errors. In other words, it learns something.

Visualize what the network learned

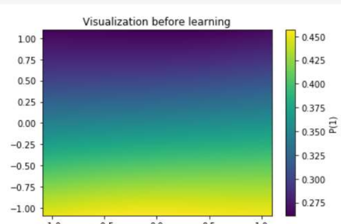
Before to see what the network learned, it would be interesting to see how the initial weights of the network would perform.

an-introduction-to-backpropagation.ipynb

File Edit View Insert Runtime Tools Help All changes saved

```python
def visualization(weights, X_data, title, superposed_training=False):
    N_test_points = 1000
    xs = np.linspace(1.1*np.min(X_data), 1.1*np.max(X_data), N_test_points)
    datapoints = np.transpose([np.tile(xs, len(xs)), np.repeat(xs, len(xs))])
    Y_initial = forward_propagation(datapoints, weights)[0].reshape(N_test_points, N_test_points)
    X1, X2 = np.meshgrid(xs, xs)
    plt.pcolormesh(X1, X2, Y_initial)
    plt.colorbar(label='P(1)')
    if superposed_training:
        plt.scatter(X_data[:N_points//2, 0], X_data[:N_points//2, 1], color='red')
        plt.scatter(X_data[N_points//2:, 0], X_data[N_points//2:, 1], color='blue')
    plt.title(title)
    plt.show()
```

```python
visualization(initial_weights, X,  'Visualization before learning')
```



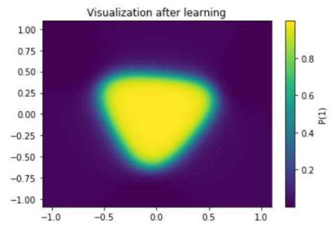
The picture above represents as a colormap the probability of a point being of class 1. As expected, the network is completely unable yet to classify correctly. Let's visualize the same thing after learning:

```python
visualization(weights, X, 'Visualization after learning')
```

an-introduction-to-backpropagation.ipynb

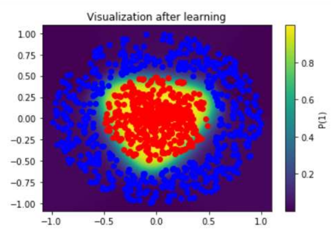
File Edit View Insert Runtime Tools Help All changes saved

```
[ ] visualization(weights, X, 'Visualization after learning')
```



Now an island of class "1" lives in the middle of the map, while the rest is of class 0. If we superimpose the training samples to this visualization we realize that our network did a pretty good job classifying this map.

```
[ ] visualization(weights, X, 'Visualization after learning', superposed_training=True)
```



DISCUSSION and CONCLUSION

The back propagation algorithm has been applied and executed successfully on a classification problem over a random generated dataset.

CRITERIA	TOTAL MARKS	MARKS OBTAINED	COMMENTS
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6		