Understanding the implementation of Neural Networks from scratch in detail

In [1]:
```
# importing required libraries
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# version of numpy library
print("Version of numpy:", np.__version__)
```

```
Version of numpy: 1.19.2
```

In [3]:
```
# version of matplotlib library
import matplotlib

print("Version of matplotlib:", matplotlib.__version__)
```

```
Version of matplotlib: 3.3.2
```

In [4]:
```
# set random seed
np.random.seed(42)
```

In [5]:
```
# creating the input array
X = np.array([[1, 0, 0, 0], [1, 0, 1, 1], [0, 1, 0, 1]])

print("Input:\n", X)

# shape of input array
print("\nShape of Input:", X.shape)
```

```
Input:
 [[1 0 0 0]
 [1 0 1 1]
 [0 1 0 1]]

Shape of Input: (3, 4)
```

In [6]:
```
# converting the input in matrix form
X = X.T
print("Input in matrix form:\n", X)

# shape of input matrix
print("\nShape of Input Matrix:", X.shape)
```

```
Input in matrix form:
 [[1 1 0]
 [0 0 1]
 [0 1 0]
 [0 1 1]]

Shape of Input Matrix: (4, 3)
```

In [7]:
```
# creating the output array
y = np.array([[1], [1], [0]])

print("Actual Output:\n", y)

# output in matrix form
y = y.T
```

```python
print("\nOutput in matrix form:\n", y)

# shape of input array
print("\nShape of Output:", y.shape)
```

```
Actual Output:
 [[1]
 [1]
 [0]]

Output in matrix form:
 [[1 1 0]]

Shape of Output: (1, 3)
```

In [8]:

```python
inputLayer_neurons = X.shape[0]  # number of features in data set
hiddenLayer_neurons = 3  # number of hidden layers neurons
outputLayer_neurons = 1  # number of neurons at output layer
inputLayer_neurons, hiddenLayer_neurons, outputLayer_neurons
```

Out[8]:

```
(4, 3, 1)
```

In [9]:

```python
# initializing weight
# Shape of weights_input_hidden should number of neurons at input layer * number of neu
weights_input_hidden = np.random.uniform(size=(inputLayer_neurons, hiddenLayer_neurons)

# Shape of weights_hidden_output should number of neurons at hidden layer * number of n
weights_hidden_output = np.random.uniform(size=(hiddenLayer_neurons, outputLayer_neuron
```

In [10]:

```python
# shape of weight matrix
weights_input_hidden.shape, weights_hidden_output.shape# We are using sigmoid as an act
                                                       # so defining the sigmoid functi
```

Out[10]:

```
((4, 3), (3, 1))
```

In [11]:

```python
# We are using sigmoid as an activation function so defining the sigmoid function here

# defining the Sigmoid Function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Forward Prapogation

In [12]:

```python
# hidden layer activations

hiddenLayer_linearTransform = np.dot(weights_input_hidden.T, X)
hiddenLayer_activations = sigmoid(hiddenLayer_linearTransform)
```

In [13]:

```python
hiddenLayer_linearTransform, hiddenLayer_activations
```

Out[13]:

```
(array([[0.37454012, 1.14069631, 1.30673106],
        [0.95071431, 1.83747495, 0.17660313],
        [0.73199394, 2.30301881, 1.12590437]]),
 array([[0.59255557, 0.75780746, 0.78696563],
        [0.72125881, 0.8626498 , 0.54403639],
        [0.67524268, 0.90912675, 0.75508227]]))
```

In [14]:

```python
# calculating the output
outputLayer_linearTransform = np.dot(weights_hidden_output.T, hiddenLayer_activations)
```

```
output = sigmoid(outputLayer_linearTransform)
```

In [15]:
```
# output
output
```

Out[15]: `array([[0.68334694, 0.72697078, 0.71257368]])`

In [16]:
```
# calculating error
error = np.square(y - output) / 2
error
```

Out[16]: `array([[0.05013458, 0.03727248, 0.25388062]])`

Backward Propagation

In [17]:
```
# rate of change of error w.r.t. output
error_wrt_output = -(y - output)
```

In [18]:
```
# error_wrt_output
error_wrt_output
```

Out[18]: `array([[-0.31665306, -0.27302922,  0.71257368]])`

In [19]:
```
# rate of change of output w.r.t. Z2
output_wrt_outputLayer_LinearTransform = np.multiply(output, (1 - output))

#output_wrt_outputLayer_LinearTransform
output_wrt_outputLayer_LinearTransform
```

Out[19]: `array([[0.2163839 , 0.19848426, 0.20481243]])`

In [20]:
```
# rate of change of Z2 w.r.t. weights between hidden and output layer
outputLayer_LinearTransform_wrt_weights_hidden_output = hiddenLayer_activations
```

In [21]:
```
# checking the shapes of partial derivatives
error_wrt_output.shape, output_wrt_outputLayer_LinearTransform.shape, outputLayer_Linea
```

Out[21]: `((1, 3), (1, 3), (3, 3))`

In [22]:
```
# shape of weights of output layer
weights_hidden_output.shape
```

Out[22]: `(3, 1)`

In [23]:
```
# rate of change of error w.r.t weight between hidden and output layer
error_wrt_weights_hidden_output = np.dot(outputLayer_LinearTransform_wrt_weights_hidden_
    (error_wrt_output * output_wrt_outputLayer_LinearTransform).T)
```

In [24]:
```
error_wrt_weights_hidden_output.shape
```

Out[24]: `(3, 1)`

In [25]:
```
# rate of change of error w.r.t. output
error_wrt_output = -(y - output)
```

In [26]:
```
# error_wrt_output
error_wrt_output
```

Out[26]:
```
array([[-0.31665306, -0.27302922,  0.71257368]])
```

In [27]:
```
# rate of change of output w.r.t. Z2
output_wrt_outputLayer_LinearTransform = np.multiply(output, (1 - output))

#output_wrt_outputLayer_LinearTransform
output_wrt_outputLayer_LinearTransform
```

Out[27]:
```
array([[0.2163839 , 0.19848426, 0.20481243]])
```

In [28]:
```
# rate of change of Z2 w.r.t. hidden layer activations
outputLayer_LinearTransform_wrt_hiddenLayer_activations = weights_hidden_output
```

In [29]:
```
# rate of change of hidden layer activations w.r.t. Z1
hiddenLayer_activations_wrt_hiddenLayer_linearTransform = np.multiply(
    hiddenLayer_activations, (1 - hiddenLayer_activations))

#hiddenLayer_activations_wrt_hiddenLayer_linearTransform
hiddenLayer_activations_wrt_hiddenLayer_linearTransform
```

Out[29]:
```
array([[0.24143347, 0.18353531, 0.16765073],
       [0.20104454, 0.11848512, 0.2480608 ],
       [0.21929   , 0.08261531, 0.18493303]])
```

In [30]:
```
# rate of change of Z1 w.r.t. weights between input and hidden layer
hiddenLayer_linearTransform_wrt_weights_input_hidden = X
```

In [31]:
```
# checking the shapes of partial derivatives
print(
    error_wrt_output.shape,
    output_wrt_outputLayer_LinearTransform.shape,
    outputLayer_LinearTransform_wrt_hiddenLayer_activations.shape,
    hiddenLayer_activations_wrt_hiddenLayer_linearTransform.shape,
    hiddenLayer_linearTransform_wrt_weights_input_hidden.shape,
)
```

```
(1, 3) (1, 3) (3, 1) (3, 3) (4, 3)
```

In [32]:
```
# shape of weights of hidden layer
weights_input_hidden.shape
```

Out[32]:
```
(4, 3)
```

In [33]:
```
# rate of change of error w.r.t weights between input and hidden layer
error_wrt_weights_input_hidden = np.dot(
    hiddenLayer_linearTransform_wrt_weights_input_hidden,
    (
        hiddenLayer_activations_wrt_hiddenLayer_linearTransform
        * np.dot(
            outputLayer_LinearTransform_wrt_hiddenLayer_activations,
            (output_wrt_outputLayer_LinearTransform * error_wrt_output),
        )
    ).T,
)
```

In [34]:
```
error_wrt_weights_input_hidden, error_wrt_weights_input_hidden.shape
```

Out[34]:
```
(array([[-0.02205044, -0.00428845, -0.00354605],
        [ 0.02036788,  0.00768731,  0.00490743],
        [-0.0082796 , -0.00136342, -0.00081405],
        [ 0.01208828,  0.00632389,  0.00409338]]),
 (4, 3))
```

In [35]:
```
# defining the learning rate
lr = 0.01
```

In [36]:
```
# initial weights_hidden_output
weights_hidden_output
```

Out[36]:
```
array([[0.83244264],
       [0.21233911],
       [0.18182497]])
```

In [37]:
```
# initial weights_input_hidden
weights_input_hidden
```

Out[37]:
```
array([[0.37454012, 0.95071431, 0.73199394],
       [0.59865848, 0.15601864, 0.15599452],
       [0.05808361, 0.86617615, 0.60111501],
       [0.70807258, 0.02058449, 0.96990985]])
```

In [38]:
```
# updating the weights of output layer
weights_hidden_output = weights_hidden_output - lr * error_wrt_weights_hidden_output
```

In [39]:
```
#weights_hidden_output
weights_hidden_output
```

Out[39]:
```
array([[0.83211079],
       [0.21250681],
       [0.18167831]])
```

In [40]:
```
# updating the weights of hidden layer
weights_input_hidden = weights_input_hidden - lr * error_wrt_weights_input_hidden

#weights_input_hidden
weights_input_hidden
```

Out[40]:
```
array([[0.37476062, 0.95075719, 0.7320294 ],
       [0.59845481, 0.15594177, 0.15594545],
       [0.05816641, 0.86618978, 0.60112315],
       [0.70795169, 0.02052126, 0.96986892]])
```

In [41]:
```
# defining the model architecture
inputLayer_neurons = X.shape[0]  # number of features in data set
hiddenLayer_neurons = 3  # number of hidden layers neurons
outputLayer_neurons = 1  # number of neurons at output layer

# initializing weight
weights_input_hidden = np.random.uniform(size=(inputLayer_neurons, hiddenLayer_neurons)
weights_hidden_output = np.random.uniform(
    size=(hiddenLayer_neurons, outputLayer_neurons)
)

# defining the parameters
lr = 0.1
epochs = 1000
```

In [42]:

```
losses = []
for epoch in range(epochs):
    ## Forward Propogation

    # calculating hidden layer activations
    hiddenLayer_linearTransform = np.dot(weights_input_hidden.T, X)
    hiddenLayer_activations = sigmoid(hiddenLayer_linearTransform)

    # calculating the output
    outputLayer_linearTransform = np.dot(
        weights_hidden_output.T, hiddenLayer_activations
    )
    output = sigmoid(outputLayer_linearTransform)

    ## Backward Propagation

    # calculating error
    error = np.square(y - output) / 2

    # calculating rate of change of error w.r.t weight between hidden and output layer
    error_wrt_output = -(y - output)
    output_wrt_outputLayer_LinearTransform = np.multiply(output, (1 - output))
    outputLayer_LinearTransform_wrt_weights_hidden_output = hiddenLayer_activations

    error_wrt_weights_hidden_output = np.dot(
        outputLayer_LinearTransform_wrt_weights_hidden_output,
        (error_wrt_output * output_wrt_outputLayer_LinearTransform).T,
    )

    # calculating rate of change of error w.r.t weights between input and hidden layer
    outputLayer_LinearTransform_wrt_hiddenLayer_activations = weights_hidden_output
    hiddenLayer_activations_wrt_hiddenLayer_linearTransform = np.multiply(
        hiddenLayer_activations, (1 - hiddenLayer_activations)
    )
    hiddenLayer_linearTransform_wrt_weights_input_hidden = X
    error_wrt_weights_input_hidden = np.dot(
        hiddenLayer_linearTransform_wrt_weights_input_hidden,
        (
            hiddenLayer_activations_wrt_hiddenLayer_linearTransform
            * np.dot(
                outputLayer_LinearTransform_wrt_hiddenLayer_activations,
                (output_wrt_outputLayer_LinearTransform * error_wrt_output),
            )
        ).T,
    )

    # updating the weights
    weights_hidden_output = weights_hidden_output - lr * error_wrt_weights_hidden_outpu
    weights_input_hidden = weights_input_hidden - lr * error_wrt_weights_input_hidden

    # print error at every 100th epoch
    epoch_loss = np.average(error)
    if epoch % 100 == 0:
        print(f"Error at epoch {epoch} is {epoch_loss:.5f}")

    # appending the error of each epoch
    losses.append(epoch_loss)
```

```
Error at epoch 0 is 0.11553
Error at epoch 100 is 0.11082
```

```
Error at epoch 200 is 0.10606
Error at epoch 300 is 0.09845
Error at epoch 400 is 0.08483
Error at epoch 500 is 0.06396
Error at epoch 600 is 0.04206
Error at epoch 700 is 0.02641
Error at epoch 800 is 0.01719
Error at epoch 900 is 0.01190
```

In [43]:
```
# updated w_ih
weights_input_hidden
```

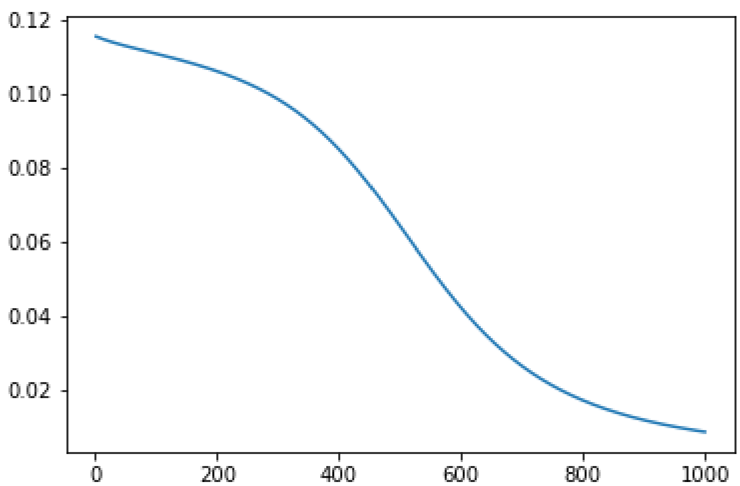
Out[43]:
```
array([[ 1.25679149,  1.72312858, -0.27336634],
       [-1.07615756, -1.73777864,  1.42316207],
       [ 0.63053865,  0.88090942, -0.03448117],
       [-0.56098781, -0.65506704,  0.61013995]])
```

In [44]:
```
# updated w_ho
weights_hidden_output
```

Out[44]:
```
array([[ 1.45176252],
       [ 2.59109536],
       [-2.18347501]])
```

In [45]:
```
# visualizing the error after each epoch
plt.plot(np.arange(1, epochs + 1), np.array(losses))
```

Out[45]:
```
[<matplotlib.lines.Line2D at 0x23d5a263220>]
```



In [46]:
```
# final output from the model
output
```

Out[46]:
```
array([[0.9155779 , 0.89643511, 0.18608711]])
```

In [47]:
```
# actual target
y
```

Out[47]:
```
array([[1, 1, 0]])
```

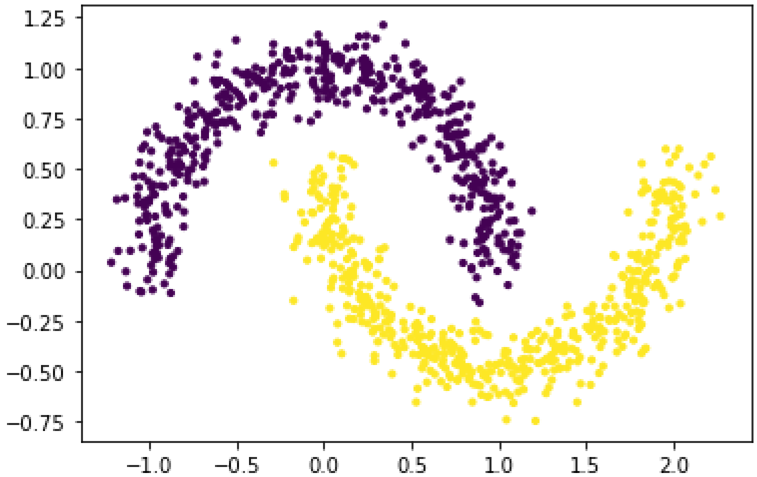
Train our model on a different dataset, and visualize the performance by plotting a decision boundary after training.

In [48]:
```
from sklearn.datasets import make_moons
```

```
X, y = make_moons(n_samples=1000, random_state=42, noise=0.1)
```

In [49]:
```
plt.scatter(X[:, 0], X[:, 1], s=10, c=y)
```

Out[49]: <matplotlib.collections.PathCollection at 0x23d5b73ef70>



In [50]:
```
X
```

Out[50]:
```
array([[-0.05146968,  0.44419863],
       [ 1.03201691, -0.41974116],
       [ 0.86789186, -0.25482711],
       ...,
       [ 1.68425911, -0.34822268],
       [-0.9672013 ,  0.26367208],
       [ 0.78758971,  0.61660945]])
```

In [51]:
```
X -= X.min()
X /= X.max()
```

In [52]:
```
X.min(), X.max()
```

Out[52]: (0.0, 1.0)

In [53]:
```
np.unique(y)
```

Out[53]: array([0, 1], dtype=int64)

In [54]:
```
X.shape, y.shape
```

Out[54]: ((1000, 2), (1000,))

In [55]:
```
X = X.T

y = y.reshape(1, -1)
```

In [56]:
```
X.shape, y.shape
```

Out[56]: ((2, 1000), (1, 1000))

In [57]:
```
# defining the model architecture
```

```
inputLayer_neurons = X.shape[0]  # number of features in data set
hiddenLayer_neurons = 10  # number of hidden layers neurons
outputLayer_neurons = 1  # number of neurons at output layer

# initializing weight
weights_input_hidden = np.random.uniform(size=(inputLayer_neurons, hiddenLayer_neurons))
weights_hidden_output = np.random.uniform(
    size=(hiddenLayer_neurons, outputLayer_neurons)
)

# defining the parameters
lr = 0.1
epochs = 10000

losses = []
for epoch in range(epochs):
    ## Forward Propogation

    # calculating hidden layer activations
    hiddenLayer_linearTransform = np.dot(weights_input_hidden.T, X)
    hiddenLayer_activations = sigmoid(hiddenLayer_linearTransform)

    # calculating the output
    outputLayer_linearTransform = np.dot(
        weights_hidden_output.T, hiddenLayer_activations
    )
    output = sigmoid(outputLayer_linearTransform)

    ## Backward Propagation

    # calculating error
    error = np.square(y - output) / 2

    # calculating rate of change of error w.r.t weight between hidden and output layer
    error_wrt_output = -(y - output)
    output_wrt_outputLayer_LinearTransform = np.multiply(output, (1 - output))
    outputLayer_LinearTransform_wrt_weights_hidden_output = hiddenLayer_activations

    error_wrt_weights_hidden_output = np.dot(
        outputLayer_LinearTransform_wrt_weights_hidden_output,
        (error_wrt_output * output_wrt_outputLayer_LinearTransform).T,
    )

    # calculating rate of change of error w.r.t weights between input and hidden layer
    outputLayer_LinearTransform_wrt_hiddenLayer_activations = weights_hidden_output
    hiddenLayer_activations_wrt_hiddenLayer_linearTransform = np.multiply(
        hiddenLayer_activations, (1 - hiddenLayer_activations)
    )
    hiddenLayer_linearTransform_wrt_weights_input_hidden = X
    error_wrt_weights_input_hidden = np.dot(
        hiddenLayer_linearTransform_wrt_weights_input_hidden,
        (
            hiddenLayer_activations_wrt_hiddenLayer_linearTransform
            * np.dot(
                outputLayer_LinearTransform_wrt_hiddenLayer_activations,
                (output_wrt_outputLayer_LinearTransform * error_wrt_output),
            )
        ).T,
    )

    # updating the weights
```

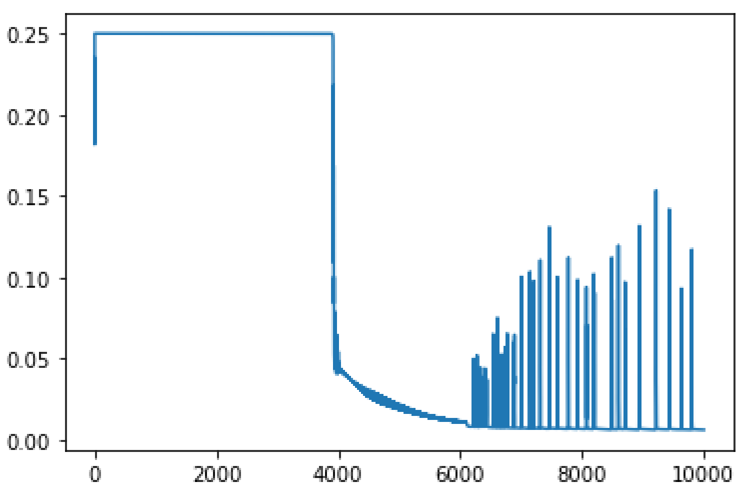
```
    weights_hidden_output = weights_hidden_output - lr * error_wrt_weights_hidden_outpu
    weights_input_hidden = weights_input_hidden - lr * error_wrt_weights_input_hidden

    # print error at every 100th epoch
    epoch_loss = np.average(error)
    if epoch % 1000 == 0:
        print(f"Error at epoch {epoch} is {epoch_loss:.5f}")

    # appending the error of each epoch
    losses.append(epoch_loss)
```

```
Error at epoch 0 is 0.23478
Error at epoch 1000 is 0.25000
Error at epoch 2000 is 0.25000
Error at epoch 3000 is 0.25000
Error at epoch 4000 is 0.05129
Error at epoch 5000 is 0.02163
Error at epoch 6000 is 0.01157
Error at epoch 7000 is 0.01110
Error at epoch 8000 is 0.00692
Error at epoch 9000 is 0.00682
```

In [58]:

```
# visualizing the error after each epoch
plt.plot(np.arange(1, epochs + 1), np.array(losses))
```

Out[58]: [<matplotlib.lines.Line2D at 0x23d5ccd0be0>]



In [59]:

```
# final output from the model
output[:, :5]
```

Out[59]:

```
array([[9.66573967e-01, 9.99646344e-01, 9.97774359e-01, 9.99484921e-01,
        1.91814527e-07]])
```

In [60]:

```
y[:, :5]
```

Out[60]:

```
array([[1, 1, 1, 1, 0]], dtype=int64)
```

In [61]:

```
# Define region of interest by data limits
steps = 1000
x_span = np.linspace(X[0, :].min(), X[0, :].max(), steps)
y_span = np.linspace(X[1, :].min(), X[1, :].max(), steps)
xx, yy = np.meshgrid(x_span, y_span)

# forward pass for region of interest
hiddenLayer_linearTransform = np.dot(
```

```
    weights_input_hidden.T, np.c_[xx.ravel(), yy.ravel()].T
)
hiddenLayer_activations = sigmoid(hiddenLayer_linearTransform)
outputLayer_linearTransform = np.dot(weights_hidden_output.T, hiddenLayer_activations)
output_span = sigmoid(outputLayer_linearTransform)

# Make predictions across region of interest
labels = (output_span > 0.5).astype(int)

# Plot decision boundary in region of interest
z = labels.reshape(xx.shape)
fig, ax = plt.subplots()
ax.contourf(xx, yy, z, alpha=0.2)

# Get predicted labels on training data and plot
train_labels = (output > 0.5).astype(int)

# create scatter plot
ax.scatter(X[0, :], X[1, :], s=10, c=y.squeeze())
```

Out[61]: <matplotlib.collections.PathCollection at 0x23d5e8226d0>