In [1]:

```
# Data science makes extensive use of linear algebra. The support for
# performing advanced linear algebra functions quickly and efficiently makes
# NumPy one of the most routinely used libraries for data science.
# To find a matrix dot product, you can use the dot() function. To find the dot
# product, the number of columns in the first matrix must match the number of
# rows in the second matrix. Here is an example.

import numpy as np
A = np.random.randn(4,5)
B = np.random.randn(5,4)
Z = np.dot(A,B)
print(Z)
```

```
[[-0.19136805 -0.14739311  1.97504063 -2.07330355]
 [-0.64645561 -0.21330557 -3.56759274  1.22695131]
 [-0.70532354 -4.1006305   2.69101909  2.19181373]
 [-4.57424439 -0.12552509 -6.70670923  3.39800888]]
```

In [2]:

```
# In addition to finding the dot product of two matrices, you can element-wise
# multiply two matrices. To do so, you can use the multiply() function.
# However, the dimensions of the two matrices must match.

import numpy as np
row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
multiply = np.multiply(nums_2d, nums_2d)
print(multiply)
```

```
[[ 100  144  169]
 [2025 1024  256]
 [2025 1024  256]]
```

In [3]:

```
# You find the inverse of a matrix via the linalg.inv() function, as shown
# below:

import numpy as np
row1 = [1,2,3]
row2 = [5,2,8]
row3 = [9,1,10]
nums_2d = np.array([row1, row2, row3])
inverse = np.linalg.inv(nums_2d)
print(inverse)
```

```
[[ 0.70588235 -1.          0.58823529]
 [ 1.29411765 -1.          0.41176471]
 [-0.76470588  1.         -0.47058824]]
```

In [4]:

```
# Similarly, the determinant of a matrix can be found using the linalg.det()
# function, as shown below:

import numpy as np
row1 = [1,2,3]
row2 = [5,2,8]
row3 = [9,1,10]
nums_2d = np.array([row1, row2, row3])
determinant = np.linalg.det(nums_2d)
print(determinant)
```

```
17.0
```

In [5]:

```
# The trace of a matrix refers to the sum of all the elements along the diagonal
# of a matrix. To find the trace of a matrix, you can use the trace() function, as
# shown below:

import numpy as np
row1 = [1,2,3]
row2 = [4,5,6]
row3 = [7,8,9]
nums_2d = np.array([row1, row2, row3])
trace = np.trace(nums_2d)
print(trace)
```

```
15
```

In [6]:

```
# Now that you know how to use the NumPy library to perform various linear
# algebra functions, let’s try to solve a system of linear equations, which is one
# of the most basic problems in linear algebra.
# A system of linear equations refers to a collection of equations with some
# unknown variables. Solving a system of linear equations refers to finding the
# values of the unknown variables in equations. One of the ways to solve a
# system of linear equations is via matrices.

# The following script creates the NumPy arrays A and B for our matrices A
# and B, respectively.

import numpy as np
A = np.array([[6, 3],[2, 4]])
B = np.array([42, 32])

# Next, the script below finds the inverse of the matrix A using the inv()
# method from the linalg submodule from the NumPy module.

A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[ 0.22222222 -0.16666667]
 [-0.11111111  0.33333333]]
```

In [7]:

```
# Finally, the script below takes the dot product of the inverse of matrix A with
# matrix B (which is a vector in our case).

X = np.dot(A_inv, B)
print(X)

# The output shows that in our system of linear equations, the values of the
# unknown variables x and y are 4 and 6, respectively.
```

```
[4. 6.]
```

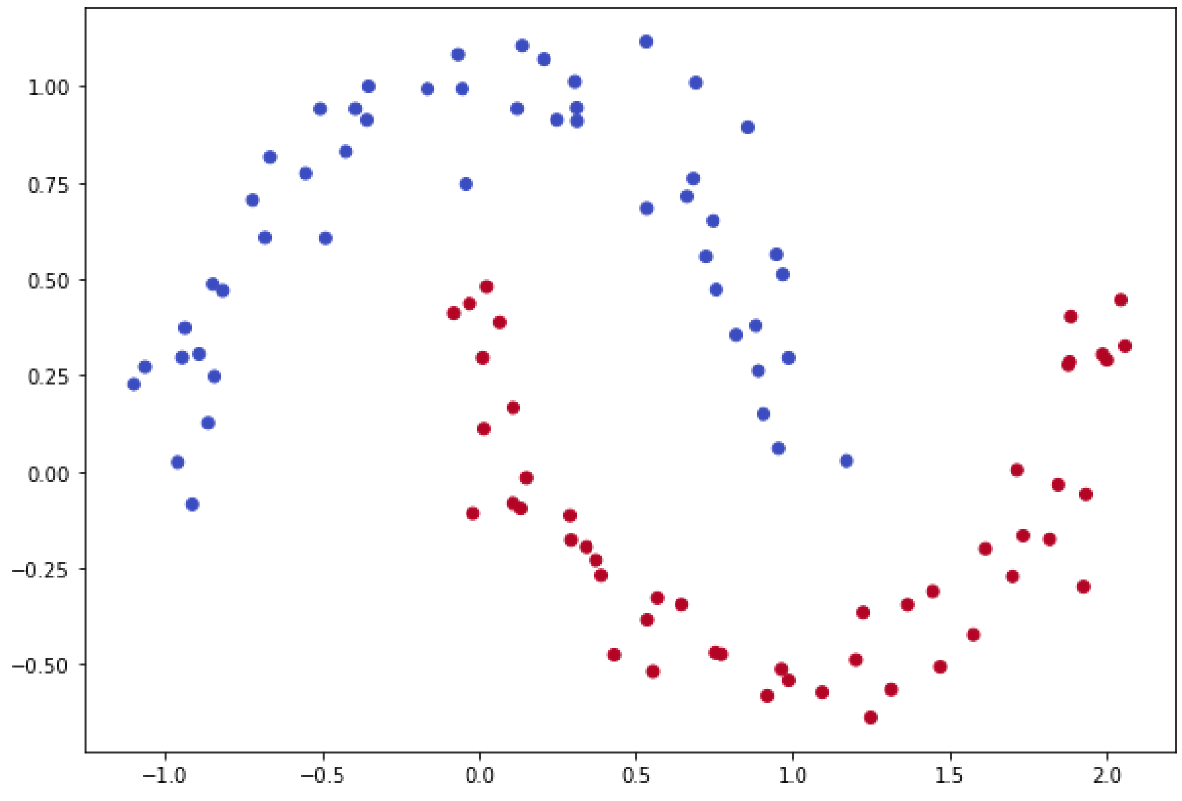
In [8]:

```
# Here, you will implement a densely connected neural network with
# a single output from scratch. In addition, you will be learning how to find a
# non-linear boundary to separate two classes.
# Let’s start by defining our dataset first:

from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(0)
X, y = datasets.make_moons(100, noise=0.10)
x1 = X[:,0]
x2 = X[:,1]
plt.figure(figsize=(10,7))
plt.scatter(x1, x2, c= y, cmap=plt.cm.coolwarm)
```

Out[8]:

```
<matplotlib.collections.PathCollection at 0x135f966b80>
```



In [9]:

```
# Our dataset has 100 records with two features and one output. You will need
# to reshape the output so that it has the same structure as the input features.

y = y.reshape(y.shape[0],1)
```

In [10]:

```
# We can now check the shape of our input features and output labels.

print(X.shape)
print(y.shape)
```

```
(100, 2)
(100, 1)
```

In [13]:

```
# In a neural network, we have an input layer, one or multiple hidden layers,
# and an output layer. In our neural network, we have two nodes in the input
# layer (since there are two features in the input), one hidden layer with four
# nodes, and one output layer with one node since we are doing binary
# classification. The number of hidden layers and the number of neurons per
# hidden layer depend upon you.

# A neural network works in two steps:
# 1. Feed Forward
# 2. BackPropagation

# In the feed forward step, the final output of a neural network is created.
# The purpose of backpropagation is to minimize the overall loss by finding the
# optimum values of weights.

# Here, you will implement the neural network. Let’s define a function that
# defines the parameters:

def define_parameters(weights):
    weight_list = []
    bias_list = []
    for i in range(len(weights) - 1):
        w = np.random.randn(weights[i], weights[i+1])
        b = np.random.randn()
    weight_list.append(w)
    bias_list.append(b)

    return weight_list, bias_list
```

In [14]:

```
# Now, since we have multiple sets of weights, the defined parameters function
# will return the list of weights connecting the input and hidden layer and the
# hidden and output layer.
# Next, we define the sigmoid function and its derivative:

def sigmoid(x):
    return1/(1+np.exp(-x))
```

In [15]:

```
def sigmoid_der(x):
    return sigmoid(x)*(1-sigmoid(x))
```

In [16]:

```
# The feed forward part of the algorithm is implemented by the prediction()
# method, as shown below:

def predictions(w, b, X):
    zh = np.dot(X,w[0]) + b[0]
    ah = sigmoid(zh)
    zo = np.dot(ah, w[1]) + b[1]
    ao = sigmoid(zo)
    return ao
```

In [17]:

```
# The following script defines the cost function:

def find_cost(ao,y):
    m = y.shape[0]
    total_cost = (1/m) * np.sum(np.square(ao - y))
    return total_cost
```

In [25]:

```
# Finally, to implement the backpropagation, we define a find_derivatives()
# function, as follows:

def find_derivatives(w, b, X):
    zh = np.dot(X,w[0]) + b[0]
    ah = sigmoid(zh)
    zo = np.dot(ah, w[1]) + b[1]
    ao = sigmoid(zo)
    # Backpropagation phase 1
    m = y.shape[0]
    dcost_dao = (1/m)*(ao-y)
    dao_dzo = sigmoid_der(zo)
    dzo_dwo = ah.T
    dwo= np.dot(dzo_dwo, dcost_dao * dao_dzo)
    dbo = np.sum(dcost_dao * dao_dzo)
    # Backpropagation phase 2
    # dcost_wh = dcost_dah * dah_dzh * dzh_dwh
    # dcost_dah = dcost_dzo * dzo_dah
    dcost_dzo = dcost_dao * dao_dzo
    dzo_dah = w[1].T
    dcost_dah = np.dot(dcost_dzo, dzo_dah)
    dah_dzh = sigmoid_der(zh)
    dzh_dwh = X.T
    dwh = np.dot(dzh_dwh, dah_dzh * dcost_dah)
    dbh = np.sum(dah_dzh * dcost_dah)
    return dwh, dbh, dwo, dbo
```

In [19]:

```python
# And to update weights by subtracting the gradient, we define the
# update_weights() function.

def update_weights(w,b,dwh, dbh, dwo, dbo, lr):
    w[0] = w[0] - lr * dwh
    w[1] = w[1] - lr * dwo
    b[0] = b[0] - lr * dbh
    b[1] = b[1] - lr * dbo
    return w, b
```

In [26]:

```python
# Here is the my_neural_network class, which is used to train the neural
# network by implementing the feed forward and backward propagation steps.

def my_neural_network(X, y, lr, epochs):
    error_list = []
    input_len = X.shape[1]
    output_len = y.shape[1]
    w,b = define_parameters([input_len, 4, output_len])
    for i in range(epochs):
        ao = predictions(w, b, X)
        cost = find_cost(ao, y)
        error_list.append(cost)
        dwh, dbh, dwo, dbo = find_derivatives (w, b, X)
        w , b = update_weights(w, b, dwh, dbh, dwo, dbo, lr)
        if i % 50 == 0 :
            print(cost)
    return w, b, error_list
```