Clean Code:

```
import numpy as np
```

```
def softmax(a) :
    e_pa = np.exp(a)
    ans = e_pa / np.sum(e_pa, axis=1, keepdims=True)
    return ans
```

```
class NeuralNetwork :

    def __init__(self, input_size, layers, output_size) :

        np.random.seed(0)

        model = {} # Dictionary

        # First Layer
        model['W1'] = np.random.randn(input_size, layers[0])
        model['b1'] = np.zeros((1, layers[0]))

        # Second Layer
        model['W2'] = np.random.randn(layers[0], layers[1])
        model['b2'] = np.zeros((1, layers[1]))

        # Third Layer
        model['W3'] = np.random.randn(layers[1], output_size)
        model['b3'] = np.zeros((1, output_size))

        self.model = model

    def forward(self, x) :

        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        b1,b2,b3 = self.model['b1'], self.model['b2'], self.model['b3']

        z1 = np.dot(x,W1) + b1
        a1 = np.tanh(z1)

        z2 = np.dot(a1,W2) + b2
        a2 = np.tanh(z2)

        z3 = np.dot(a2,W3) + b3
        y_ = softmax(z3)

        self.activation_outputs = (a1, a2, y_)
        return y_

    def backward(self, x, y, learning_rate=0.001) :

        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        b1,b2,b3 = self.model['b1'], self.model['b2'], self.model['b3']
        m = x.shape[0]

        a1, a2, y_ = self.activation_outputs
```

Within Neural Network class

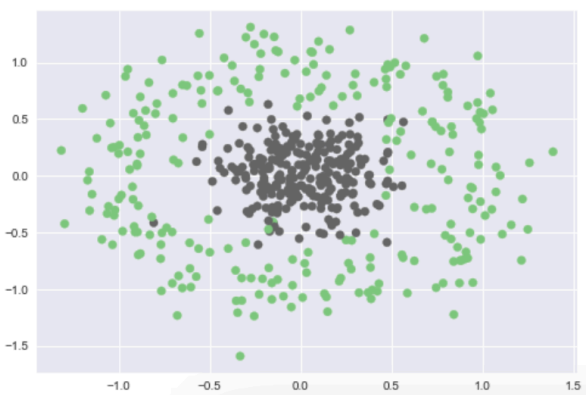
```
    def backward(self, x, y, learning_rate=0.001) :

        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        b1,b2,b3 = self.model['b1'], self.model['b2'], self.model['b3']
        m = x.shape[0]

        a1, a2, y_ = self.activation_outputs

        delta3 = y_ - y
        dw3 = np.dot(a2.T, delta3)
        db3 = np.sum(delta3, axis=0)

        delta2 = (1-np.square(a2)) * np.dot(delta3,W3.T)
        dw2 = np.dot(a1.T, delta2)
        db2 = np.sum(delta2, axis=0)

        delta1 = (1-np.square(a1)) * np.dot(delta2,W2.T)
        dw1 = np.dot(X.T, delta1)
        db1 = np.sum(delta1, axis=0)

        # Update the Model Parameters using Gradient Descent
        self.model["W1"] -= learning_rate * dw1
        self.model["b1"] -= learning_rate * db1

        self.model["W2"] -= learning_rate * dw2
        self.model["b2"] -= learning_rate * db2

        self.model["W3"] -= learning_rate * dw3
        self.model["b3"] -= learning_rate * db3

    def predict(self, x) :
        y_out = self.forward(x)
        return np.argmax(y_out, axis=1)

    def summary(self) :
        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        a1,a2,y_ = self.activation_outputs

        print("W1 ", W1.shape)
        print("A1 ", a1.shape)

        print("W2 ", W2.shape)
        print("A2 ", a2.shape)

        print("W3 ", W3.shape)
        print("Y_ ", y_.shape)
```

```
def loss(y_oht, p) :
    l = -np.mean(y_oht * np.log(p))
    return l ;

def one_hot(y, depth) :

    m = y.shape[0]
    y_oht = np.zeros((m,depth))
    y_oht[np.arange(m), y] = 1

    return y_oht
```

```
## Generate Dataset
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
```

```
X,Y = make_circles(n_samples=500, shuffle=True, noise=.2, random_state=1, factor=0.2)
```

```
plt.style.use("seaborn")
plt.scatter(X[:,0], X[:,1], c=Y, cmap=plt.cm.Accent)
plt.show()
```



Training Our Model

```
model = NeuralNetwork(input_size=2, layers=[10,5], output_size=2)
```

2 layers: 10 neurons in 1st layer and 5 neurons in 2nd layer

```
model.forward(X[0])
```

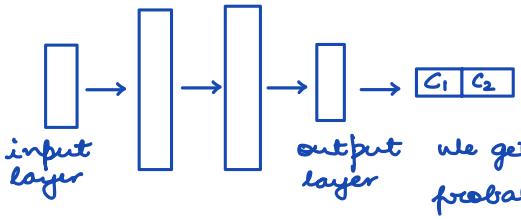
```
array([[0.52335135, 0.47664865]])
```

there are 2 output classes

```
model.forward(X)
```

```
[0.33181008, 0.66818992],
[0.42029777, 0.57970223],
[0.39580255, 0.60419745],
[0.46748259, 0.53251741],
[0.56851961, 0.43148039],
[0.48633496, 0.51366504],
[0.63608743, 0.36391257],
[0.4463488 , 0.5536512 ],
[0.43329275, 0.56670725],
[0.6181356 , 0.3818644 ],
[0.396852  , 0.603148  ],
[0.43234738, 0.56765262],
[0.71176096, 0.28823904],
[0.37243677, 0.62756323],
[0.59305146, 0.40694854],
[0.43534634, 0.56465366],
[0.51067345, 0.48932655],
[0.5736724 , 0.4263276 ],
[0.5642692 , 0.4357308 ]])
```

if we pass one example, we get 2 probabilities.



we get two probability one for C1 and other for C2

```
model.forward(X).shape
```

```
(500, 2)
```

If we pass m examples, we get m x 2 matrix.

```
model.summary()
```

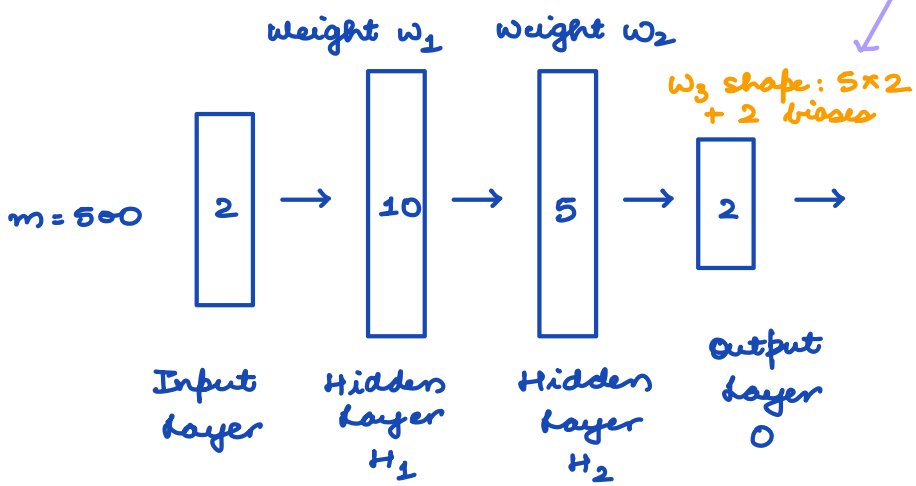
```
W1  (2, 10)
A1  (500, 10)
W2  (10, 5)
A2  (500, 5)
W3  (5, 2)
Y_  (500, 2)
```

outputs produced by H1 for 500 examples each of the 10 unit will produce some output.

Total Parameter

W_1 Shape : 2x10 + 10 Biases

W_2 Shape : 10x5 + 5 Biases



Output : 500x2 500x10 500x5 500x2

- Earlier data was represented in 2 dimension.
- Then it is represented in a higher dimensional space i.e. 10 dimension.
- Then it is brought down to 5 dimension.
- Then finally brought down to 2 dimension where each no. represents probability of 2 class.

- Actually neural network is trying to take the data from one space to other high dimensional space and then bring it down to small dimensional space.

```
def train(X, Y, model, epochs, learning_rate, logs=True) :
    training_loss = []

    classes = 2
    Y_OHT = one_hot(Y, classes)

    for ix in range(epochs) :

        Y_ = model.forward(X)
        l = loss(Y_OHT, Y_)
        training_loss.append(l)
        model.backward(X, Y_OHT, learning_rate)

        if(logs) :
            print("Epoch %d Loss %.4f"%(ix,l))


    return training_loss
```

→ add the loss in list

```
losses = train(X, Y, model, 500, 0.001)
```

```
Epoch 481 Loss 0.0396
Epoch 482 Loss 0.0396
Epoch 483 Loss 0.0396
Epoch 484 Loss 0.0395
Epoch 485 Loss 0.0395
Epoch 486 Loss 0.0395
Epoch 487 Loss 0.0395
Epoch 488 Loss 0.0395
Epoch 489 Loss 0.0395
Epoch 490 Loss 0.0395
Epoch 491 Loss 0.0395
Epoch 492 Loss 0.0395
Epoch 493 Loss 0.0394
Epoch 494 Loss 0.0394
Epoch 495 Loss 0.0394
Epoch 496 Loss 0.0394
Epoch 497 Loss 0.0394
Epoch 498 Loss 0.0394
Epoch 499 Loss 0.0394
```

```
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```