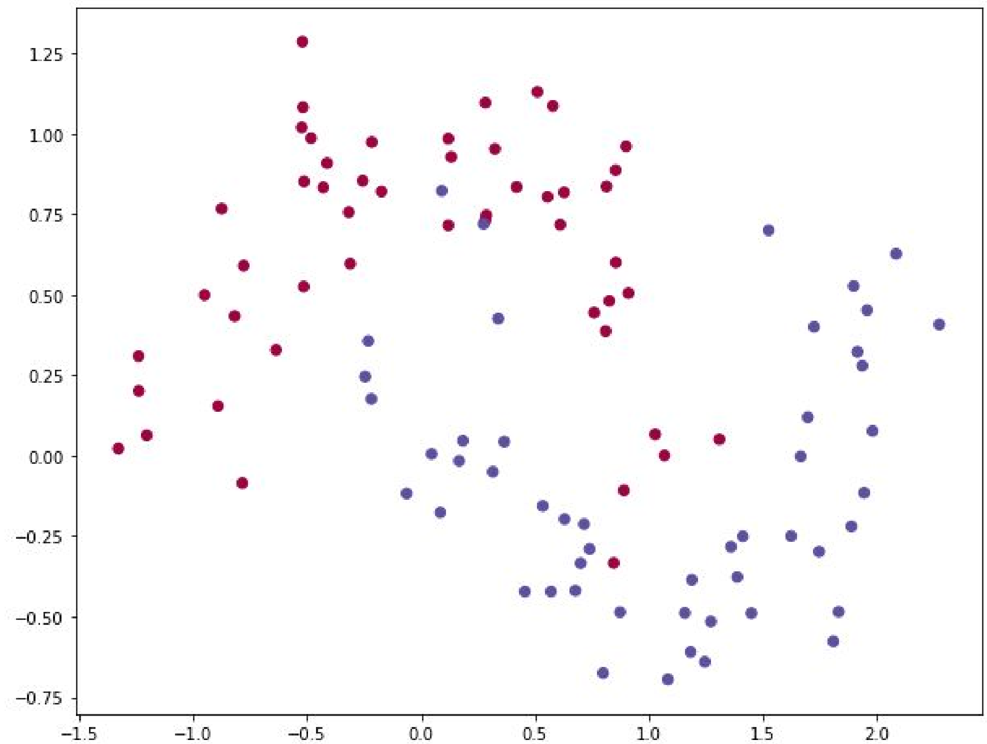
Example implementation of ANN using numpy:

We can have one input, one hidden layer and an output layer:

Steps involved in implementing an ANN:

- We take an input matrix ‘X’ and an output matrix ‘y’



The blue dots represent one class and red dots represent another class of points (Can be applied to a scenario)

- Taking some random weight and and bias -(this will later be set in the program during backpropagation) using random function in numpy.

```
weight_to_hiddenlayer=np.random.uniform(size=(<number of neurons in the input layer>,<number of neurons in the hidden layer>))
weight_outputlayer=np.random.uniform(size=(<number of neurons in the hidden layer>,<number of neurons in the output layer>))
```

The same should be done for the bias.

- Forward Propagation:

  y = wX + b

  Here, y is the output value, X is the input value, w is the weight and b is the bias.

  Try the forward propagation removing bias and understand why we need to include it.

  Linear Transformation:

  Perform matrix dot product to the inputs and the weights assigned to the edges and then add biases of the hidden layer.

  hiddenL_input = matrix_dot_product(X,weight_to_hiddenlayer) + bias_to_hiddenlayer

  Non-Linear Transformation:

  Apply activation function. Here use sigmoid. You can try using other activation functions and differentiate the output. Actually, activation function should be used based on the problem.

  hiddenL_activations = sigmoid(hiddenL_input)

  Linear Transformation for the output layer:

  *outputL_input = matrix_dot_product (hiddenL_activations * weight_outputlayer) + bias_outputlayer*

  Then apply sigmoid to the output layer's input:

  output = sigmoid(outputL_input)

- Back Propagation:

  Compare the predicted output and actual output.Calculate the gradient of error:

  Error = (Actual - Predicted) → E = (y - output)

$$\frac{ds(x)}{dx} = \frac{1}{1+e^{-x}}$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 \frac{d}{dx}(1+e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 e^{-x}(-1)$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)(-e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{-e^{-x}}{1+e^{-x}}\right)$$

$$= s(x)(1-s(x))$$

  Calculate the gradient of sigmoid function (Derivative of sigmoid) x *(1-x):

outputL_slope = derivatives_sigmoid(output)

hiddenL_slope = derivatives_sigmoid(hiddenL_activations)

- Compute change factor(delta) at output layer, dependent on the gradient of error multiplied by the slope of output layer activation

  *d_output = E * outputL_slope*
- Calculate the error at hidden layer

  Error_at_hidden_layer = matrix_dot_product(d_output, weight_outputlayer.Transpose)
- Compute change factor (delta)

  *d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer*
- Update the weights in the network from the errors calculated

  *weight_outputlayer = weight_outputlayer + matrix_dot_product(hiddenlayer_activations.Transpose, d_output)*learning_rate*

  *weight_hiddenlayer = weight_hiddenlayer + matrix_dot_product(X.Transpose,d_hiddenlayer)*learning_rate*
- In the same way update the bias

  bias at output_layer = bias at output_layer + sum of delta of output_layer at row-wise * learning_rate

  bias at hidden_layer =bias at hidden_layer + sum of delta of output_layer at row-wise * learning_rate

  *bias_hiddenL = bias_hiddenL + sum(d_hiddenlayer, axis=0, keepdims = True) * learning_rate*

  *bias_outputL = bias_outputL + sum(d_output, axis=0, keepdims = True)*learning_rate*