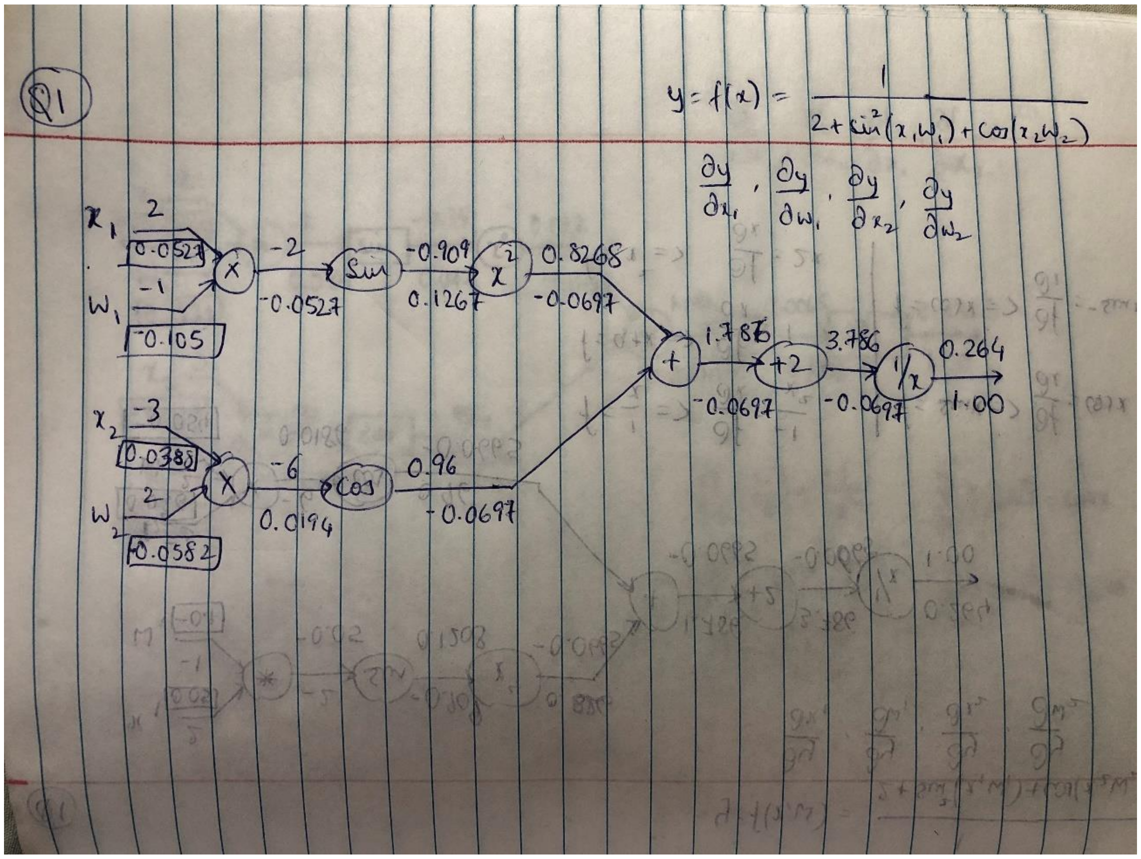
Intro to Deep Learning

Assignment 2

1. **Practice of scalar-based backpropagation**

   a. **Manual Calculation**

   The computation graph along with the intermediate calculations are shown in the figure below:

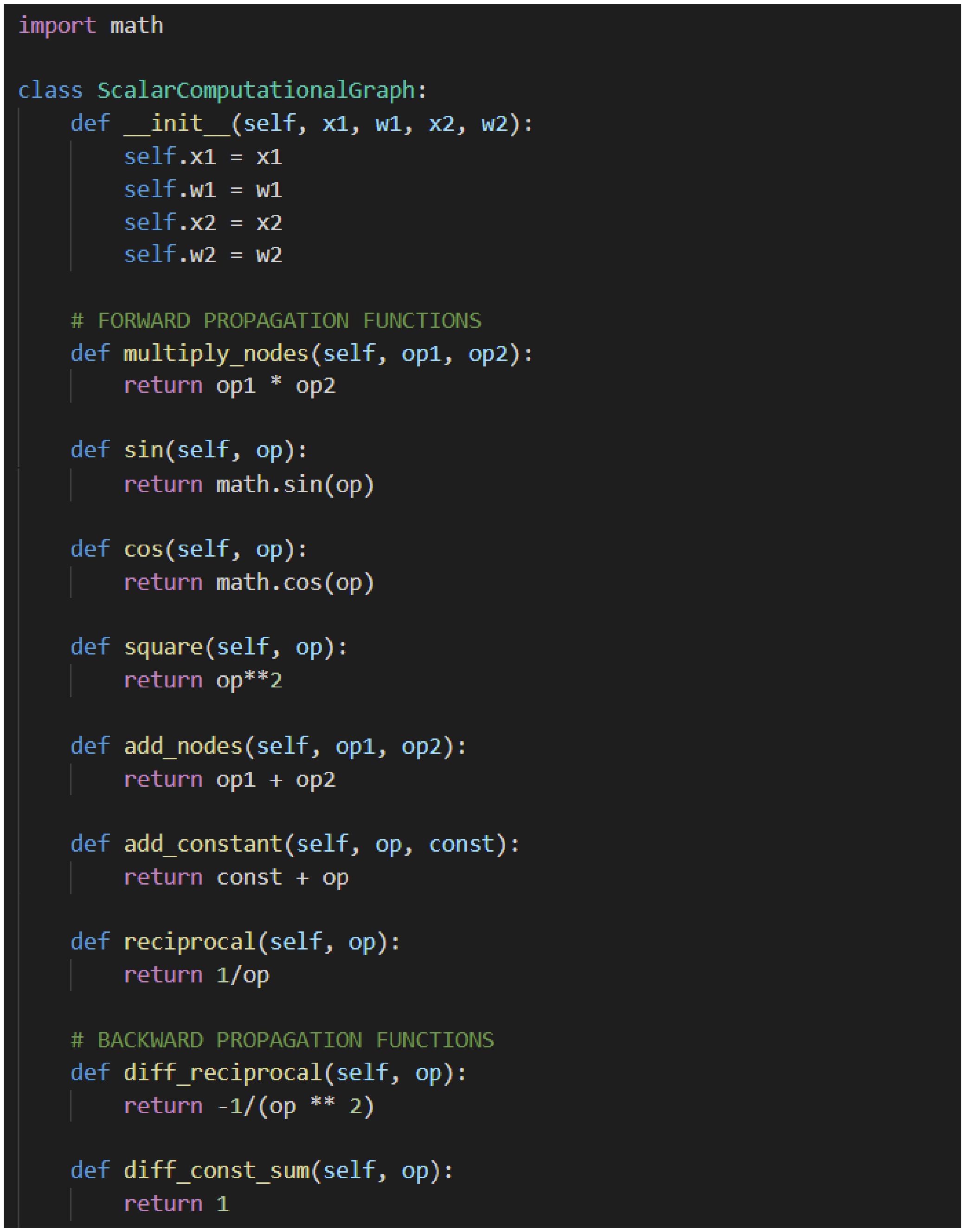


Q1
$y = f(x) = \frac{1}{2+\sin^2(x_1 w_1) + \cos(x_2 w_2)}$
$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial w_1}, \frac{\partial y}{\partial x_2}, \frac{\partial y}{\partial w_2}$
x_1 2
0.0527
w_1 -1
-0.105
-2
-0.0527
sin
-0.909
0.1267
x^2
0.8268
-0.0697
x_2 -3
0.0388
w_2 2
-0.0582
-6
0.0194
cos
0.96
-0.0697
+
1.786
-0.0697
+2
3.786
-0.0697
1/x
0.264
1.00

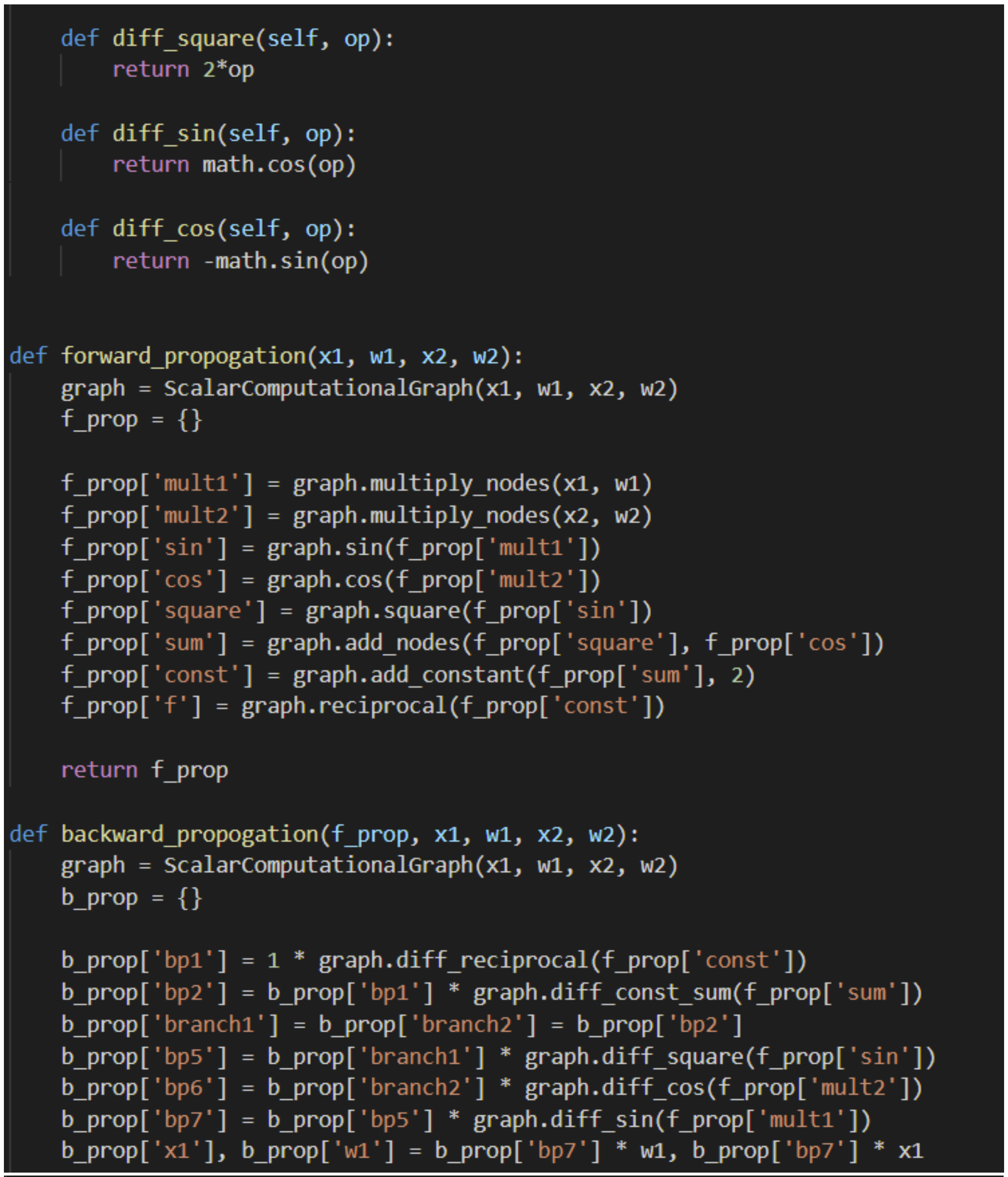
The values shown above the arrows between each node in the graph represent the values obtained by forward propagation and those below represent the values obtained by back propagation. The values in the boxes below x1, x2, w1 and w2 represent the final gradient values ($\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}$) obtained by back propagation.

b. Program to verify the calculation.

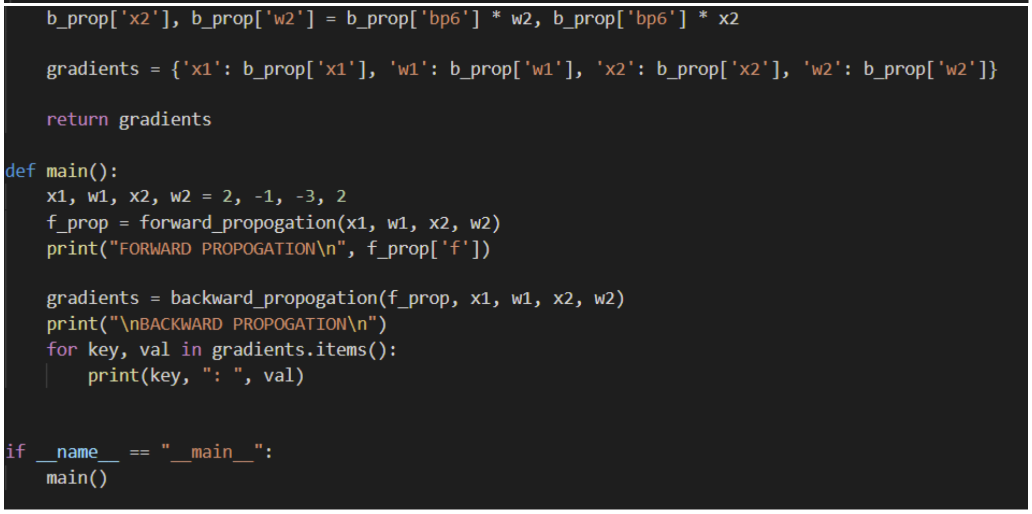
The code used to verify the above result is shown in the screenshots below:



```python
import math

class ScalarComputationalGraph:
    def __init__(self, x1, w1, x2, w2):
        self.x1 = x1
        self.w1 = w1
        self.x2 = x2
        self.w2 = w2

    # FORWARD PROPAGATION FUNCTIONS
    def multiply_nodes(self, op1, op2):
        return op1 * op2

    def sin(self, op):
        return math.sin(op)

    def cos(self, op):
        return math.cos(op)

    def square(self, op):
        return op**2

    def add_nodes(self, op1, op2):
        return op1 + op2

    def add_constant(self, op, const):
        return const + op

    def reciprocal(self, op):
        return 1/op

    # BACKWARD PROPAGATION FUNCTIONS
    def diff_reciprocal(self, op):
        return -1/(op ** 2)

    def diff_const_sum(self, op):
        return 1
```



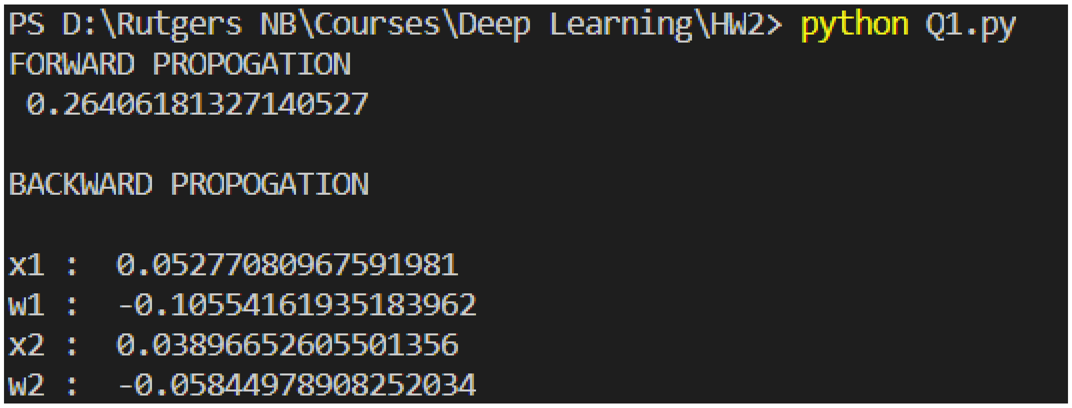
```python
    def diff_square(self, op):
        return 2*op

    def diff_sin(self, op):
        return math.cos(op)

    def diff_cos(self, op):
        return -math.sin(op)


def forward_propogation(x1, w1, x2, w2):
    graph = ScalarComputationalGraph(x1, w1, x2, w2)
    f_prop = {}

    f_prop['mult1'] = graph.multiply_nodes(x1, w1)
    f_prop['mult2'] = graph.multiply_nodes(x2, w2)
    f_prop['sin'] = graph.sin(f_prop['mult1'])
    f_prop['cos'] = graph.cos(f_prop['mult2'])
    f_prop['square'] = graph.square(f_prop['sin'])
    f_prop['sum'] = graph.add_nodes(f_prop['square'], f_prop['cos'])
    f_prop['const'] = graph.add_constant(f_prop['sum'], 2)
    f_prop['f'] = graph.reciprocal(f_prop['const'])

    return f_prop

def backward_propogation(f_prop, x1, w1, x2, w2):
    graph = ScalarComputationalGraph(x1, w1, x2, w2)
    b_prop = {}

    b_prop['bp1'] = 1 * graph.diff_reciprocal(f_prop['const'])
    b_prop['bp2'] = b_prop['bp1'] * graph.diff_const_sum(f_prop['sum'])
    b_prop['branch1'] = b_prop['branch2'] = b_prop['bp2']
    b_prop['bp5'] = b_prop['branch1'] * graph.diff_square(f_prop['sin'])
    b_prop['bp6'] = b_prop['branch2'] * graph.diff_cos(f_prop['mult2'])
    b_prop['bp7'] = b_prop['bp5'] * graph.diff_sin(f_prop['mult1'])
    b_prop['x1'], b_prop['w1'] = b_prop['bp7'] * w1, b_prop['bp7'] * x1
```



```python
    b_prop['x2'], b_prop['w2'] = b_prop['bp6'] * w2, b_prop['bp6'] * x2

    gradients = {'x1': b_prop['x1'], 'w1': b_prop['w1'], 'x2': b_prop['x2'], 'w2': b_prop['w2']}

    return gradients

def main():
    x1, w1, x2, w2 = 2, -1, -3, 2
    f_prop = forward_propogation(x1, w1, x2, w2)
    print("FORWARD PROPOGATION\n", f_prop['f'])

    gradients = backward_propogation(f_prop, x1, w1, x2, w2)
    print("\nBACKWARD PROPOGATION\n")
    for key, val in gradients.items():
        print(key, ": ", val)


if __name__ == "__main__":
    main()
```

To solve this problem, I used a class to initialize the computation graph with the initial values for x1, x2, w1 and w2. I also used the class to define the functions that would calculate the forward as well as back propagation gradients. After defining these, I created two functions, one for forward propagation and for back propagation, that would compute the gradients using the class functions for all nodes in the computation graph. I used dictionaries to store the intermediate gradient values so that it was convenient for me to verify the calculations at each stage.

The output obtained on running the code is shown below:



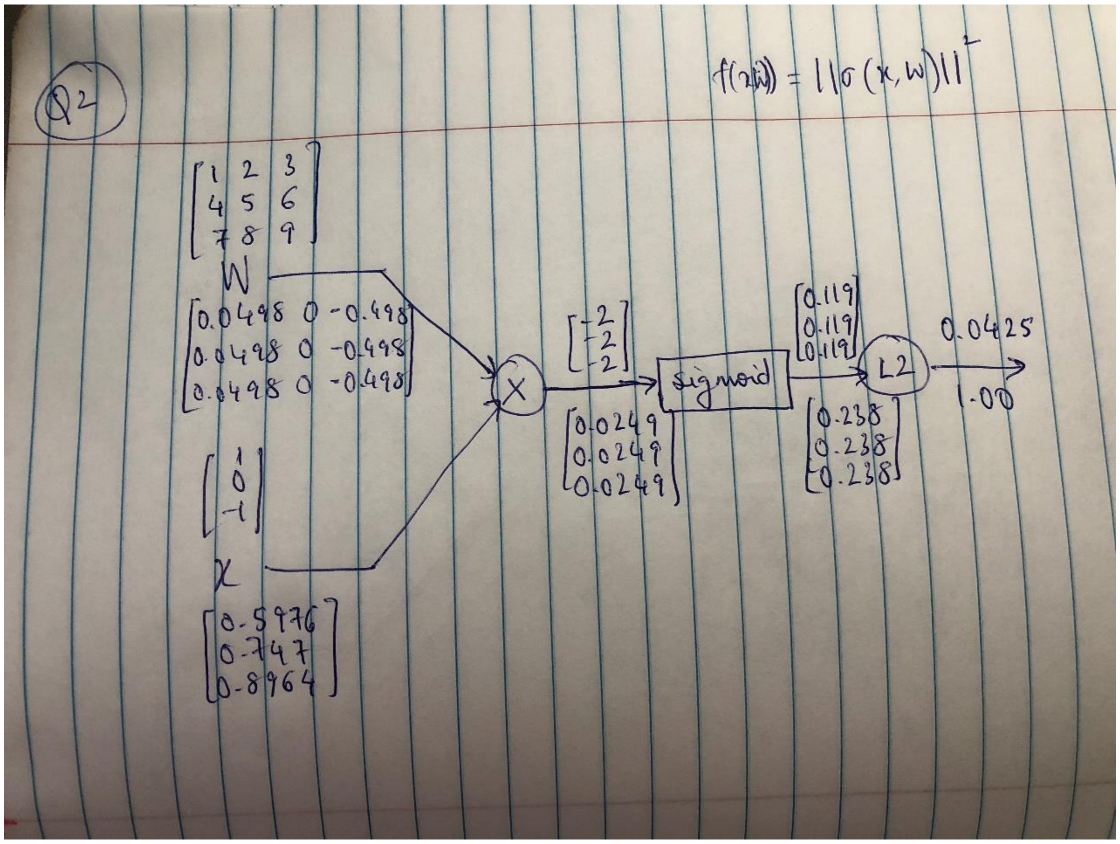
```
PS D:\Rutgers NB\Courses\Deep Learning\HW2> python Q1.py
FORWARD PROPOGATION
 0.26406181327140527

BACKWARD PROPOGATION

x1 :  0.05277080967591981
w1 :  -0.10554161935183962
x2 :  0.03896652605501356
w2 :  -0.05844978908252034
```

These values match what I had obtained by manual calculation, so the result is verified.

2. Practice of vector-based backpropagation

a. Manual Calculation

The computation graph along with the intermediate calculations are shown in the figure below:

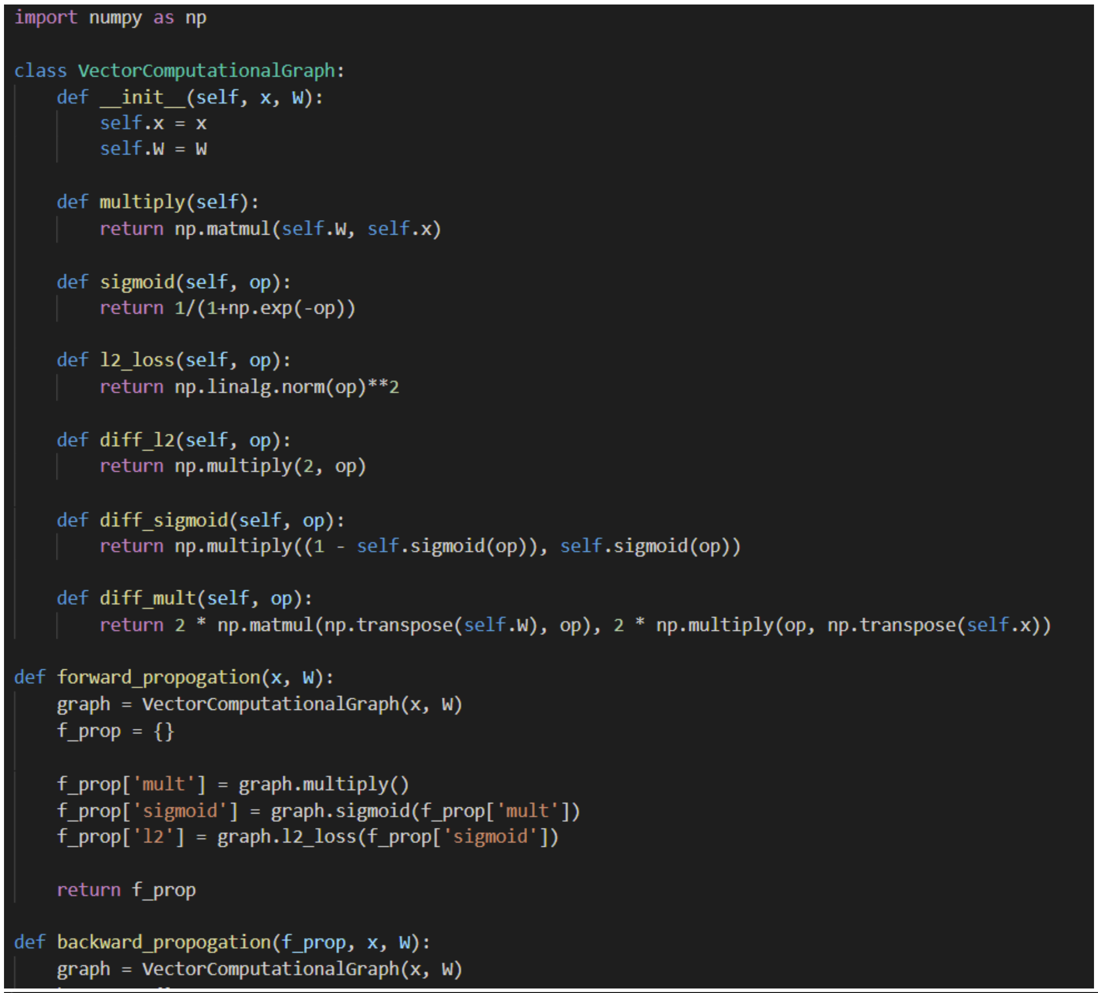


Q2
$f(x,W) = ||\sigma(x,W)||^2$
$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}$
W
$\begin{bmatrix}0.0498&0&-0.498\\0.0498&0&-0.498\\0.0498&0&-0.498\end{bmatrix}$
$\begin{bmatrix}1\\0\\-1\end{bmatrix}$
x
$\begin{bmatrix}0.5976\\0.747\\0.8964\end{bmatrix}$
×
$\begin{bmatrix}-2\\-2\\-2\end{bmatrix}$
$\begin{bmatrix}0.0249\\0.0249\\0.0249\end{bmatrix}$
sigmoid
$\begin{bmatrix}0.119\\0.119\\0.119\end{bmatrix}$
$\begin{bmatrix}0.238\\0.238\\0.238\end{bmatrix}$
L2
0.0425
1.00

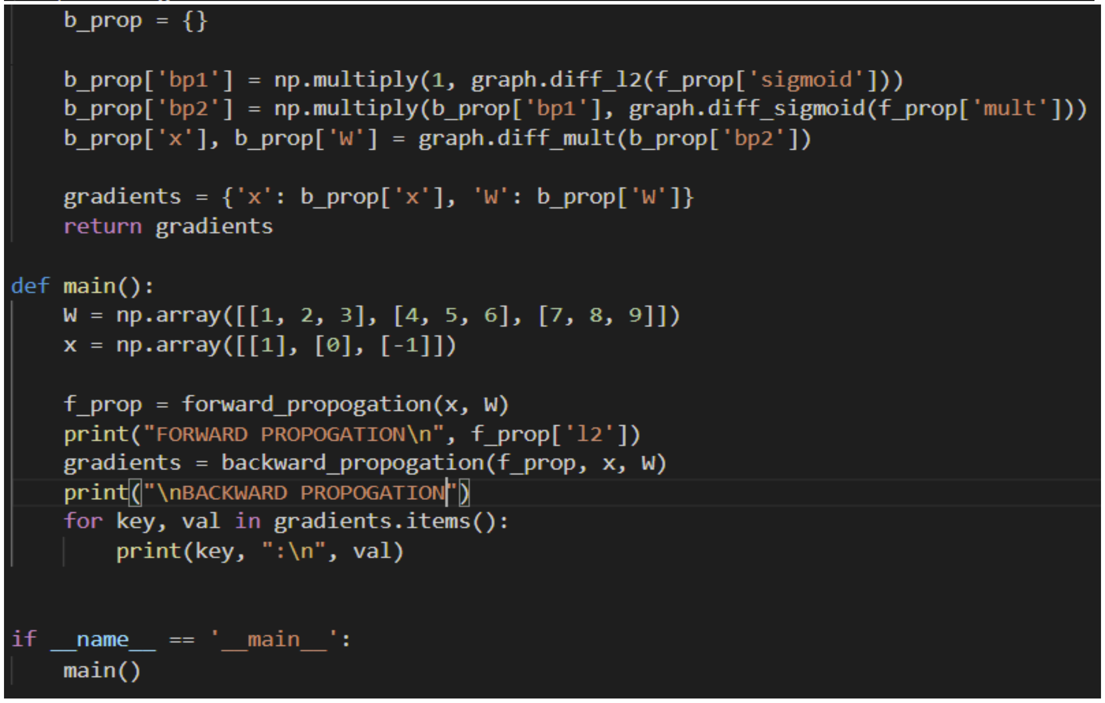
The values shown above the arrows between each node in the graph represent the values obtained by forward propagation and those below represent the values obtained by back propagation. The values in the boxes below x and W represent the final gradient values ($\frac{\partial f}{\partial x}, \frac{\partial f}{\partial W}$) obtained by back propagation.

b. Program to Verify the Calculation

The code used to verify the above result is shown in the screenshots below:



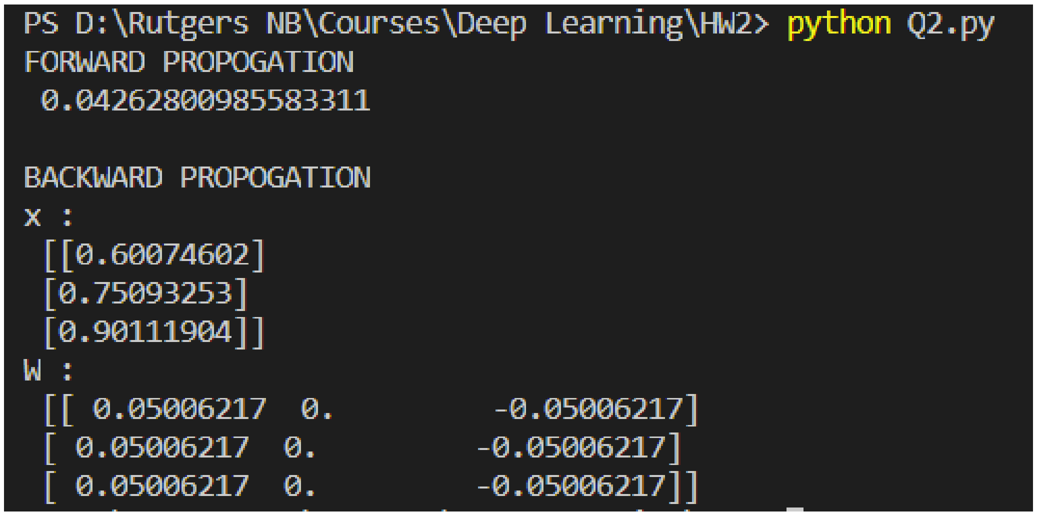
```python
import numpy as np

class VectorComputationalGraph:
    def __init__(self, x, W):
        self.x = x
        self.W = W

    def multiply(self):
        return np.matmul(self.W, self.x)

    def sigmoid(self, op):
        return 1/(1+np.exp(-op))

    def l2_loss(self, op):
        return np.linalg.norm(op)**2

    def diff_l2(self, op):
        return np.multiply(2, op)

    def diff_sigmoid(self, op):
        return np.multiply((1 - self.sigmoid(op)), self.sigmoid(op))

    def diff_mult(self, op):
        return 2 * np.matmul(np.transpose(self.W), op), 2 * np.multiply(op, np.transpose(self.x))

def forward_propogation(x, W):
    graph = VectorComputationalGraph(x, W)
    f_prop = {}

    f_prop['mult'] = graph.multiply()
    f_prop['sigmoid'] = graph.sigmoid(f_prop['mult'])
    f_prop['l2'] = graph.l2_loss(f_prop['sigmoid'])

    return f_prop

def backward_propogation(f_prop, x, W):
    graph = VectorComputationalGraph(x, W)
```



```python
    b_prop = {}

    b_prop['bp1'] = np.multiply(1, graph.diff_l2(f_prop['sigmoid']))
    b_prop['bp2'] = np.multiply(b_prop['bp1'], graph.diff_sigmoid(f_prop['mult']))
    b_prop['x'], b_prop['W'] = graph.diff_mult(b_prop['bp2'])

    gradients = {'x': b_prop['x'], 'W': b_prop['W']}
    return gradients

def main():
    W = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    x = np.array([[1], [0], [-1]])

    f_prop = forward_propogation(x, W)
    print("FORWARD PROPOGATION\n", f_prop['l2'])
    gradients = backward_propogation(f_prop, x, W)
    print("\nBACKWARD PROPOGATION")
    for key, val in gradients.items():
        print(key, ":\n", val)


if __name__ == '__main__':
    main()
```

Like the first question, I created a class to initialize the computation graph with the input and weight vectors as well as defined functions to compute the intermediate values. I also used dictionaries to store the intermediate calculations for the gradients and to simplify the matrix operations involved in the code, I used numpy functions to compute the gradients.

The output obtained on running the code is shown below:



```
PS D:\Rutgers NB\Courses\Deep Learning\HW2> python Q2.py
FORWARD PROPOGATION
 0.04262800985583311

BACKWARD PROPOGATION
x :
 [[0.60074602]
 [0.75093253]
 [0.90111904]]
W :
 [[ 0.05006217  0.          -0.05006217]
 [ 0.05006217  0.          -0.05006217]
 [ 0.05006217  0.          -0.05006217]]
```

The results obtained match with the manually computed results, so the result is verified.