COMP30230 Connectionist Computing – Programming Assignment

A multi layer perceptron is made up of four main stages

Stage 1: Initializing Network Parameters – randomly initiate the weights and set the biases to zero.

```
def __init__(self, inputSize, hiddenSizes, outputSize):  cill *
    self.inputSize = inputSize
    self.hiddenSizes = hiddenSizes
    self.outputSize = outputSize
    self.number_of_layers = len(hiddenSizes) + 1

    self.weights = []
    self.biases = []
    sizes = [inputSize] + hiddenSizes + [outputSize]
    for i in range(1, self.number_of_layers + 1):
        self.weights.append(np.random.randn(sizes[i], sizes[i-1]))
        self.biases.append(np.random.randn(sizes[i], 1))
```

Stage 2: Forward Propagation – apply activation functions to compute the activations.

```
def forward(self, X):  6 usages  cill *
    self.activations = [X]
    self.z = []
    for i in range(self.number_of_layers):
        z = np.dot(self.weights[i], self.activations[i]) + self.biases[i]
        self.z.append(z)
        if i < self.number_of_layers - 1:
            a = self.tanh(z)
        else:
            a = z
        self.activations.append(a)
    return self.activations[-1]
```

Stage 3: Backward Propagation – adjust the weights and biases using the gradient descent algorithm.

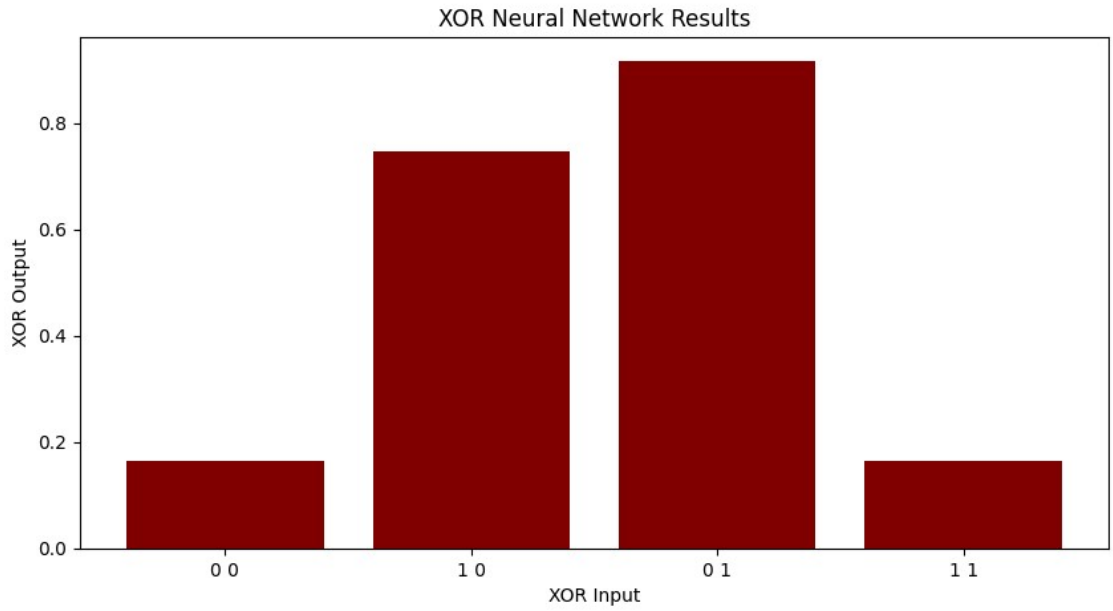
```
def backward(self, X, y):  3 usages  cill *
    m = X.shape[1]
    gradients = []
    dZ = self.activations[-1] - y
    for i in range(self.number_of_layers - 1, -1, -1):
        dW = (1 / m) * np.dot(dZ, self.activations[i].T)
        db = (1 / m) * np.sum(dZ, axis=1, keepdims=True)
        gradients.append((dW, db))

        if i > 0:
            dA = np.dot(self.weights[i].T, dZ)
            dZ = dA * self.gradient_tanh(self.z[i - 1])

    return gradients[::-1]
```

Stage 4: Update parameters – update the weights and biases.

```
def update_parameters(self, gradients, learning_rate):  3 usages  cill *
    # Update parameters using gradients and learning rate
    for i in range(self.number_of_layers):
        self.weights[i] -= learning_rate * gradients[i][0]
        self.biases[i] -= learning_rate * gradients[i][1]
```

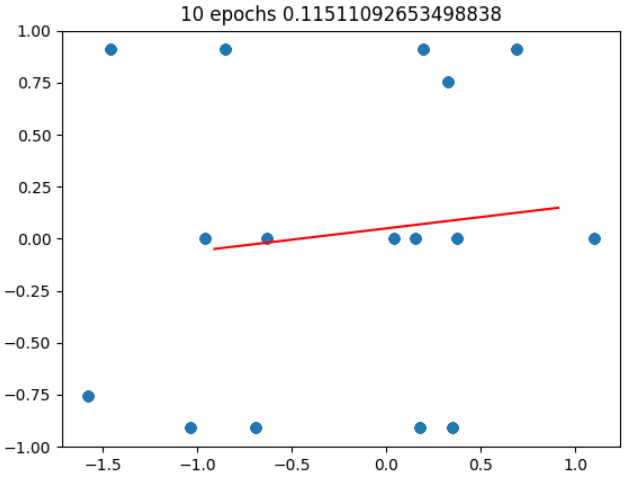
Test 1:

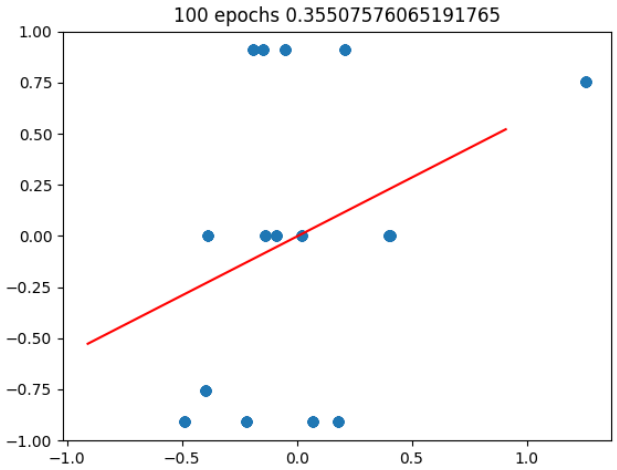
```
if __name__ == "__main__":
    arr = np.array([[0,0], [1,0], [0,1], [0,0]])
    arr2 = np.array([[0], [1], [1], [0]])
    average = (0,0,0,0)

    for tests in range(10):
        mlp=MLP( inputSize: 2,  hiddenSizes: [3, 3], outputSize: 1)
        for epoch in range(500):
            outputs = mlp.forward(arr.T)
            gradients = mlp.backward(arr.T, arr2.T)
            mlp.update_parameters(gradients,  learning_rate: 0.01)
            loss = np.mean((outputs - arr2.T)**2)

        test_output = mlp.forward(arr.T)
        average = average + test_output

    print(average/10)
```



The XOR neural networks don’t produce perfect results but it does resemble the general pattern we are looking for which is 0 0 = 0, 1 0 = 1, 0 1 = 1, 1 1 = 0. And which more epochs it would become closer to the desired result (as we will see later).
A potential solution to this would be to use integer casting, in the example above setting anything above 0.5 to 1 and below 0.5 to 0 would give us the desired results, and because we are want to have a binary output this wouldn’t lead to any loss in information.

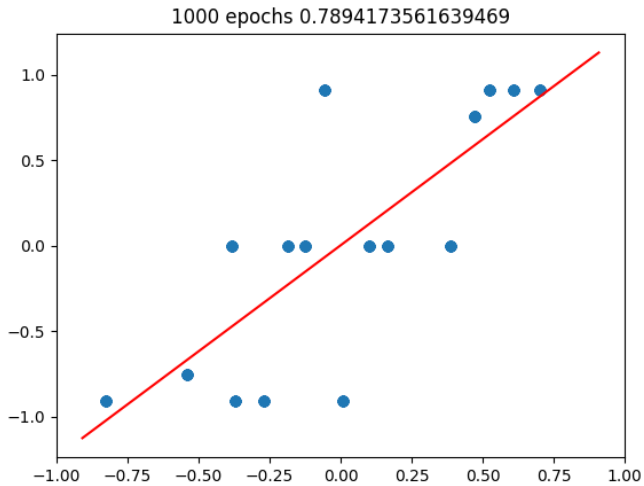
Test 2:

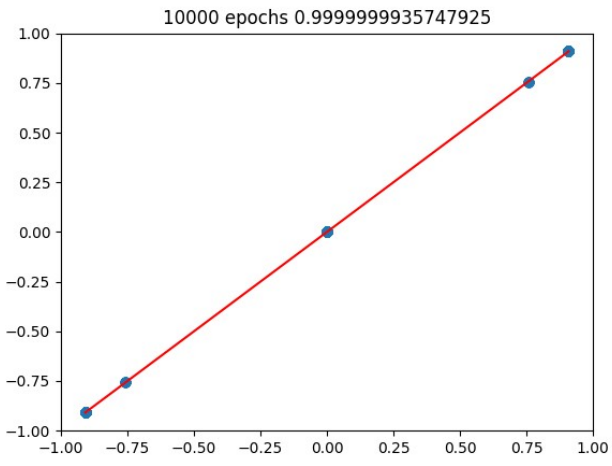


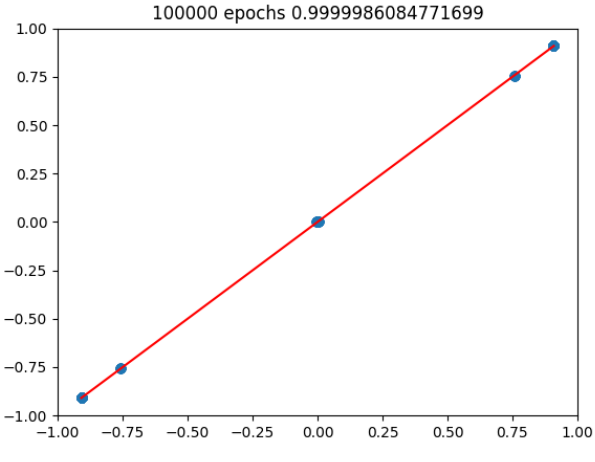



```python
if __name__ == "__main__":
    x = random.choice( a: [1, -1], size=(500, 4))
    arr = []
    for loop in range(500):
        arr.append(math.sin(x[loop][0] - x[loop][1] + x[loop][2] - x[loop][3]))

    vectorInput=np.array(x)
    vectorOutput=np.array(arr)
    mlp=MLP( inputSize: 4,  hiddenSizes: [5,5],  outputSize: 1)

    for epoch in range(400):
        outputs = mlp.forward(vectorInput.T)
        gradients = mlp.backward(vectorInput.T, vectorOutput.T)
        mlp.update_parameters(gradients,  learning_rate: 0.01)
        loss = np.mean((outputs-vectorOutput.T)**2)
        print(f"Epoch {epoch+1} - Loss: {loss}")

    test_output = mlp.forward(vectorInput.T)
    test_loss = np.mean((test_output - vectorOutput.T)**2)
    print(f"Test Loss: {test_loss}")
```

When I first ran this test my initial thought was that due to the output of sin being potentially more complicated than XOR the correlation between arr and the test_output was low.
But as I increased the amount of epochs up to 10,000 the correlation became much stronger, which shows the importance of running a lot of tests.
But that can become resource intensive, with the last test here taking a few seconds to run.



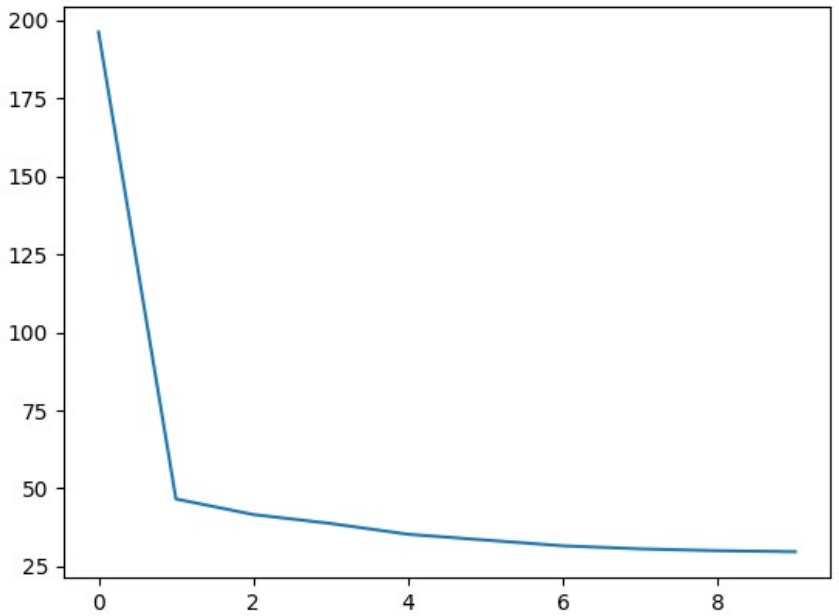




```python
def char_to_int(char):  1 usage  new *
    return ord(char) - ord('A')+1

if __name__ == "__main__":
    letters = []
    numbers = []
    numbers2=[[]]
    results=[]
    counter=0

    try:
        with open('data.txt', 'r') as file:
            for line in file:
                elements = line.strip().split(',')
                letters.append(char_to_int(elements[0]))
                numbers.extend(map(int, elements[1:]))
                numbers2.append(numbers)
                numbers=[]
    except FileNotFoundError:
        print("File not found")

    numbers2=numbers2[1:]
    vectorInput=np.array(numbers2)
    vectorOutput=np.array(letters)

    mlp=MLP( inputSize: 16,  hiddenSizes: [10,10],  outputSize: 1)

    for epoch in range(1000):
        outputs = mlp.forward(vectorInput.T)
        gradients = mlp.backward(vectorInput.T, vectorOutput.T)
        mlp.update_parameters(gradients,  learning_rate: 0.01)
        loss = np.mean((outputs-vectorOutput.T)**2)
        if (epoch%100==0):
            results.append(loss)

    test_output = mlp.forward(vectorInput.T)
    test_loss = np.mean((test_output - vectorOutput.T) ** 2)
    plt.plot(results)
    plt.show()
```

Test 3:

And while increasing the number of epochs does make our neural network more accurate as we can see from the graph below is does give us diminishing returns.
Indicating that if we are getting diminishing returns for exponentially increasing running time, it might be best to put some thought into what the ideal number of epochs is instead of just increasing them blindly.



Conclusion

I learned a lot during this assignment, I originally intended on doing the project in C++ because I have never written anything serious in python before, but everywhere online said that python is the best language for neural networks so I decided to use the best tool for the job, and in doing so I learned quite a bit about the syntax and libraries of python which I was quite unfamiliar about.

I also learned how to set up the code for a multi-layered perceptron of course, I would still need to do more work to fully understand how these perceptrons work and how to practically apply them. But I was able to create working prototype which produced interesting results in the testing.

Each test explores something slightly different, there is no ideal way to run these perceptrons, having too many epochs will lead to a higher run time and not having enough will lead to inaccurate results, but if we accept that are results are going to be inaccurate and have something such as integer casting in place to deal with that it mightn't be a problem.