Exercise project X

Based on my thoughts, I want to explain how a neural network works.

In the very first step, I imported the basic libraries and then created two functions. The first one is the ReLU activation function, which returns a positive number; otherwise, it returns 0.

```python
def activation_ReLu(number):
   if number > 0:
       return number
   else:
       return 0
```

The second function will compute the partial derivative of the RelU. It will return 1 for positive otherwise 0 input.

```python
def activation_ReLu_partial_derivative(number):
   if number > 0:
       return 1
   else:
       return 0
```

Then, we generate a dataset of 100 training samples by looping 100 times. There are two variables, `n1` and `n2`. `n1` stores values between 0 and 4, while `n2` stores values between 3 and 6. Using a formula, we calculate the value of `n3`, convert it into an integer, append all the variables as a list, and return the result.

```python
def generate_train_data():
   result = []
   for x in range(100):
       n1 = np.random.randint(0, 5)
       n2 = np.random.randint(3, 7)
       n3 = n1 ** 2 + n2 + np.random.randint(0, 5)
       n3 = int(n3)
       result.append([n1, n2, n3])
   return result
```

Then, I provide weights with initial value, which is will change during the training time which is w1 to w6. Along with this I also initilized the biases and store the original weights' values with biases. After that, I start to set training parameters which is Learning Rate (RL) with value 0.005 along with epoch with value 1000. Its means the number of times the training process will loop through the dataset. There is a dataset for training with dimensional arrays. Also, the array of loss value points which is being use for track the model performance.

```
w1 = 1
w2 = 0.5
w3 = 1
w4 = -0.5
w5 = 1
w6 = 1
bias1 = 0.5
bias2 = 0
bias3 = 0.5
w1_initial = w1
w2_initial = w2
w3_initial = w3
w4_initial = w4
w5_initial = w5
w6_initial = w6
bias1_initial = bias1
bias2_initial = bias2
bias3_initial = bias3
LR = 0.005
epochs = 1000
data = [
  [1, 0, 2],
  [2, 1, 6],
  [3, 3, 17]
]
data = generate_train_data()
loss_points = []
```

Now, extracting the features from datasets along with targeted output value, which is n3 These values and fed into a function for the calculation of the loss errors and weights as well.

```
for row in data:
        input1 = row[0] # first feature
        input2 = row[1]  # second feature
        true_value = row[2] # true value
```

Now, calculate the output of Node 1, which is a hidden layer neuron.
Then, the neuron applies a weighted sum of the inputs (input1 and input2) using the weights w1 and w3.
Along with this bias term (bias1) is added to shift the activation threshold.
The result is passed through the ReLU activation function (activation_ReLu()), which ensures non-linearity.

```
# NODE 1 OUTPUT
        node_1_output = input1 * w1 + input2 * w3 + bias1
        node_1_output = activation_ReLu(node_1_output)
```

This is similar to Node 1 but with different weights and bias (w2, w4, and bias2).
It computes another weighted sum of the same inputs (input1, input2).
The result is passed through ReLU activation.

```
# NODE 2 OUTPUT
        node_2_output = input1 * w2 + input2 * w4 + bias2
        node_2_output = activation_ReLu(node_2_output)
```

Unlike Nodes 1 and 2, this node does not take input1 and input2 directly.
Instead, it takes the outputs of Node 1 and Node 2.
The weighted sum of node_1_output and node_2_output is calculated using w5 and w6, with bias3 added.

Finally, ReLU activation is applied again.

```
# NODE 3 OUTPUT
        # we can just use Node 1 and 2 outputs, since they
        # already contain the the previous weights
        node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
        node_3_output = activation_ReLu(node_3_output)
```

Now Node_3_output is the final output of the neural network after passing through all the layers.
This value is treated as the predicted output of the model.

```
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
```

Now, I am going to implement backpropagation, starting from the last layer. It calculates the gradient of the loss with respect to w5 and updates its value.

```
 deriv_L_w5 = 2 * node_1_output * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value) new_w5 = w5 - LR * deriv_L_w5
```

Here I am going to update the w6 using gradient descent, just like w5 but now focusing on the contribution of Node 2 to the error.

```
  # solve w6 and update the new value
  deriv_L_w6 = 2 * node_2_output * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value) new_w6 = w6 - LR * deriv_L_w6
```

Unlike weights, biases are not multiplied by input values. The update step is similar to w5 and w6 but without any node multiplications.
Biases help shift the activation function, allowing the network to learn better.

```
      # solve bias3 and update the new value
       deriv_L_b3 = 2 * 1 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)new_b3 = bias3 - LR * deriv_L_b3
```

Now, we are moving into backpropagation for the first layer, which requires using the chain rule because w1 affects the final loss indirectly through multiple layers.

Here, w5 and w6, which directly contribute to the final output node 3 the first-layer weight w1 contributes indirectly by influencing node 1 which then affects node 3. So, we must use the chain rule to compute the gradient.

```
deriv_L_w1_left = 2 * w5 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
       deriv_L_w1_right = activation_ReLu_partial_derivative(input1 * w1 + input2 * w3
+ bias1) * input1
       deriv_L_w1 = deriv_L_w1_left * deriv_L_w1_right
       new_w1 = w1 - LR * deriv_L_w1
```

The update for w2 is computed using the chain rule, where the first term deriv_L_w2_left captures how changes in w2 affect the final loss through node 2 and node 3, while the second term deriv_L_w2_right accounts for the effect of w2 on node 2. Considering the ReLU activation function's derivative, the gradient descent step then adjusts w2. to reduce the error.

```
# weight 2
       deriv_L_w2_left = 2 * w6 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
       deriv_L_w2_right = activation_ReLu_partial_derivative(input1 * w2 + input2 * w4
+ bias2) * input1
       deriv_L_w2 = deriv_L_w2_left * deriv_L_w2_right
       new_w2 = w2 - LR * deriv_L_w2
```

The update for w3 is computed using the chain rule, where the first term deriv_L_w3_left captures how changes in w3 affect the final loss through node 1 and node 3.
while the second term deriv_L_w3_right accounts for the effect of w3 on node 1.considering the ReLU activation function's derivative, the gradient descent step then adjusts w3.

```
# weight 3
       deriv_L_w3_left = 2 * w5 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
       deriv_L_w3_right = activation_ReLu_partial_derivative(input1 * w1 + input2 * w3
+ bias1) * input2
       deriv_L_w3 = deriv_L_w3_left * deriv_L_w3_right
       new_w3 = w3 - LR * deriv_L_w3
```

The update for w4 is calculated using the chain rule. The first part, deriv_L_w4_left, shows how changes in w4 affect the final loss through Node 2 and Node 3. The second part, deriv_L_w4_right, captures how w4 influences Node 2, considering the ReLU activation's derivative. Finally, gradient descent adjusts w4 to reduce the error.

```
# weight 4
       deriv_L_w4_left = 2 * w6 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
       deriv_L_w4_right = activation_ReLu_partial_derivative(input1 * w2 + input2 * w4
+ bias2) * input2
       deriv_L_w4 = deriv_L_w4_left * deriv_L_w4_right
       new_w4 = w4 - LR * deriv_L_w4
```

Then for bias 1 is computed using the chain rule, where the first term deriv_L_b1_left captures how changes in bias 1 affect the final loss through node 1 and node 3.
 while the second term deriv_L_b1_right accounts for the effect of bias 1 on node 1.considering the ReLU activation function's derivative, the gradient descent step then adjusts bias 1 to reduce the errors.

```
# bias 1
        deriv_L_b1_left = 2 * w5 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
        deriv_L_b1_right = activation_ReLu_partial_derivative(input1 * w1 + input2 * w3
+ bias1) * 1
        deriv_L_b1 = deriv_L_b1_left * deriv_L_b1_right
        new_b1 = bias1 - LR * deriv_L_b1
```

Then going to update for bias 2 is computed using the chain rule, where the first term deriv_L_b2_left captures how changes in bias 2 affect the final loss through node 2 and node 3.
while there is second term deriv_L_b2_right accounts for the effect of bias 2 on node 2.considering the ReLU activation function's derivative, the gradient descent step then adjusts bias 2.

```
# bias 2
        deriv_L_b2_left = 2 * w6 * (node_1_output * w5 + node_2_output * w6 + bias3 -
true_value)
        deriv_L_b2_right = activation_ReLu_partial_derivative(input1 * w2 + input2 * w4
+ bias2) * 1
        deriv_L_b2 = deriv_L_b2_left * deriv_L_b2_right
        new_b2 = bias2 - LR * deriv_L_b2
```

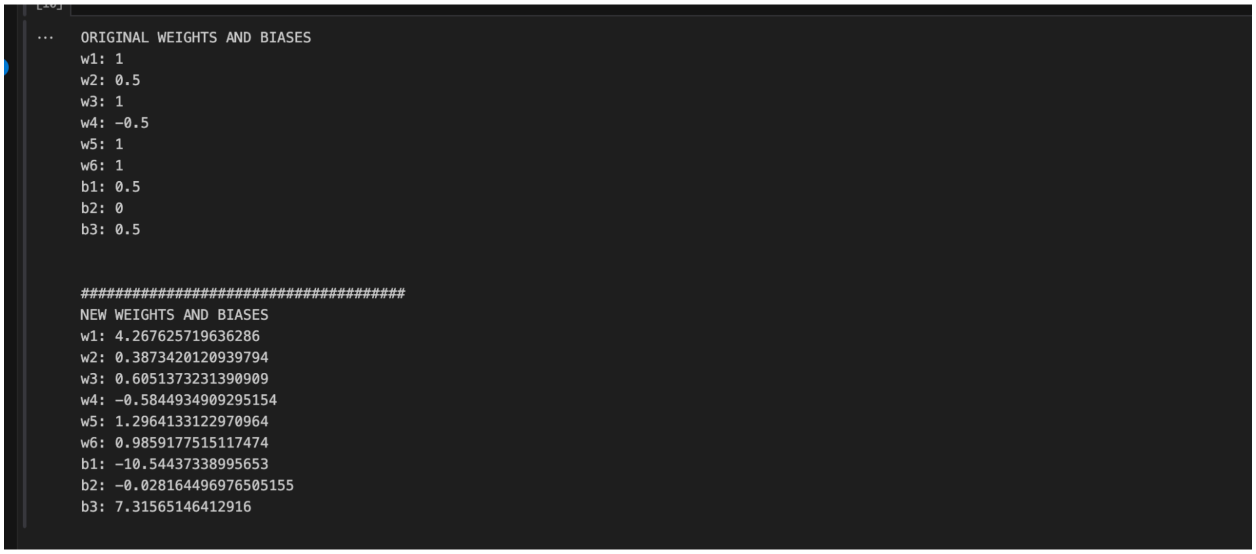
Finally, the weights and biases are updated with their new values after the backpropagation and gradient descent steps. The new values of w1, w2, w3, w4, w5, w6, bias1, bias2, and bias3 are stored in their respective variables. effectively adjusting the model parameters to minimize the loss and improve the model's predictions for the next iteration.

```
#FINALLY UPDATE THE EXISTING WEIGHTS!
        w1 = new_w1
        w2 = new_w2
        w3 = new_w3
        w4 = new_w4
        w5 = new_w5
        w6 = new_w6
        bias1 = new_b1
        bias2 = new_b2
        bias3 = new_b3
```

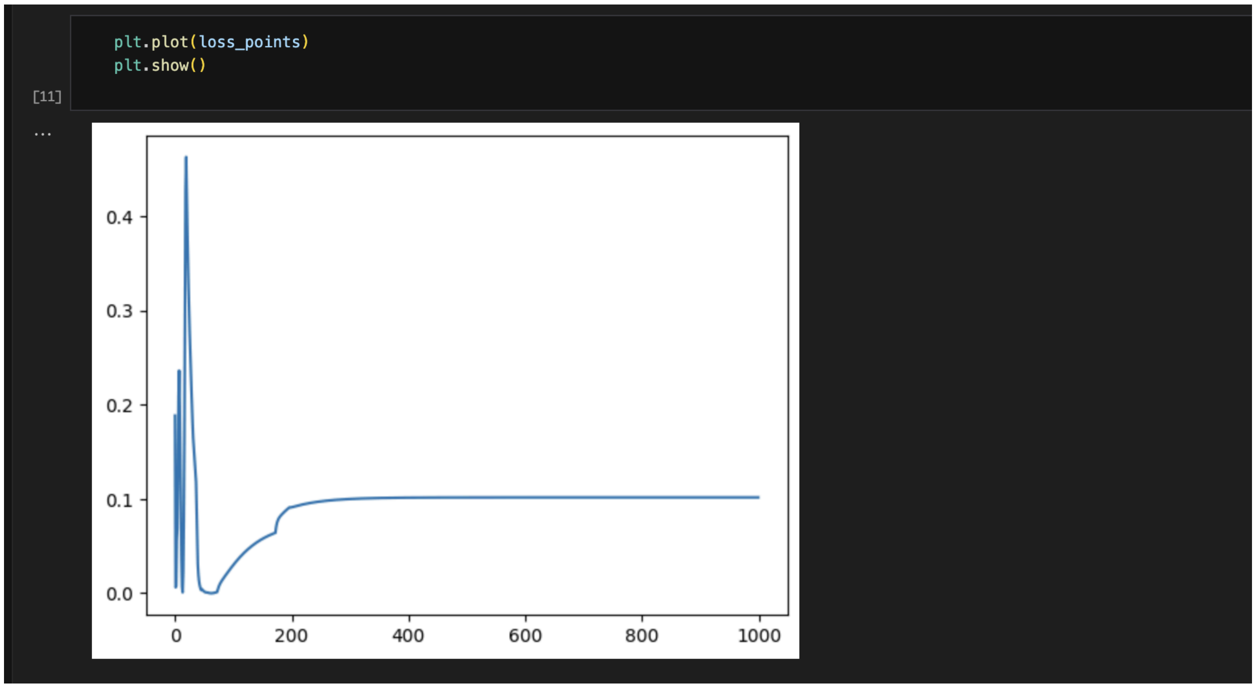
```
loss_points.append(loss)
   print(f"Epoch: {epoch +1}, loss {loss}")
```

```
Epoch: 998, loss 0.10165885792982245
Epoch: 999, loss 0.10165885800698403
Epoch: 1000, loss 0.10165885808272854
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings…



```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -0.5
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


######################################
NEW WEIGHTS AND BIASES
w1: 4.267625719636286
w2: 0.3873420120939794
w3: 0.6051373231390909
w4: -0.5844934909295154
w5: 1.2964133122970964
w6: 0.9859177515117474
b1: -10.54437338995653
b2: -0.028164496976505155
b3: 7.31565146412916
```



```
plt.plot(loss_points)
plt.show()
```

[11]

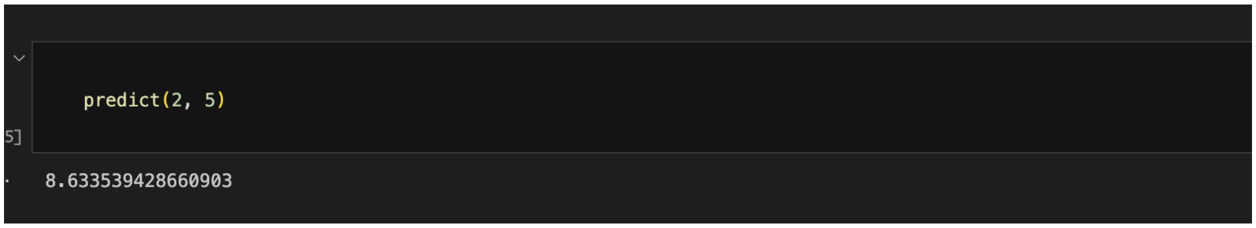
Predict function for Forward Pass.

The predicting function takes two inputs, performs a forward pass through the neural network using the current weights and biases, and returns the output from the final node after applying the ReLU activation function.

```
def predict(x1, x2):
   input1 = x1
   input2 = x2

   # FORWARD PASS

   # NODE 1 OUTPUT
   node_1_output = input1 * w1 + input2 * w3 + bias1
   node_1_output = activation_ReLu(node_1_output)

   # NODE 2 OUTPUT
   node_2_output = input1 * w2 + input2 * w4 + bias2
   node_2_output = activation_ReLu(node_2_output)

   # NODE 3 OUTPUT
   # we can just use Node 1 and 2 outputs, since they
   # its already contain the the previous weights
   node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
   node_3_output = activation_ReLu(node_3_output)
```

```
return node_3_output
```

The function will take two inputs, n1 and n2, and return the predicted value of the neural network for those inputs.
The function will take 2 as n1 and 5 as n2, and return the predicted value of the neural network for those inputs.

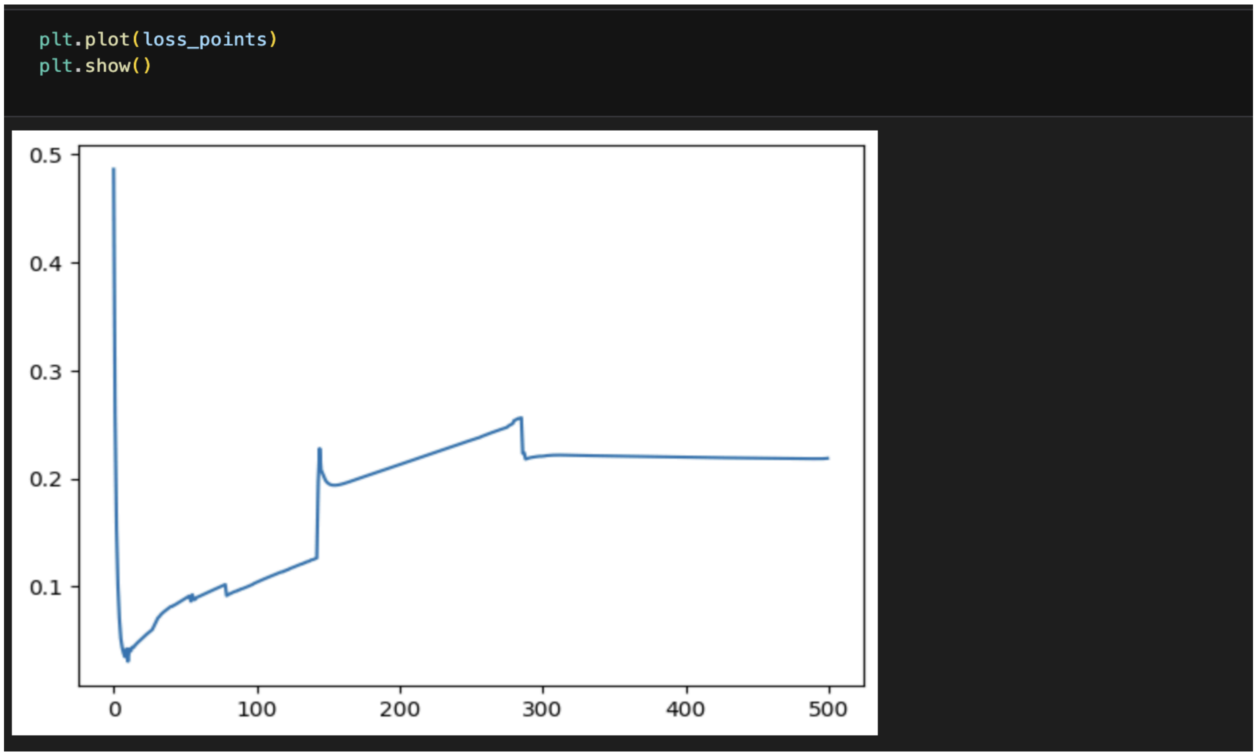


```
predict(2, 5)
```

```
8.633539428660903
```

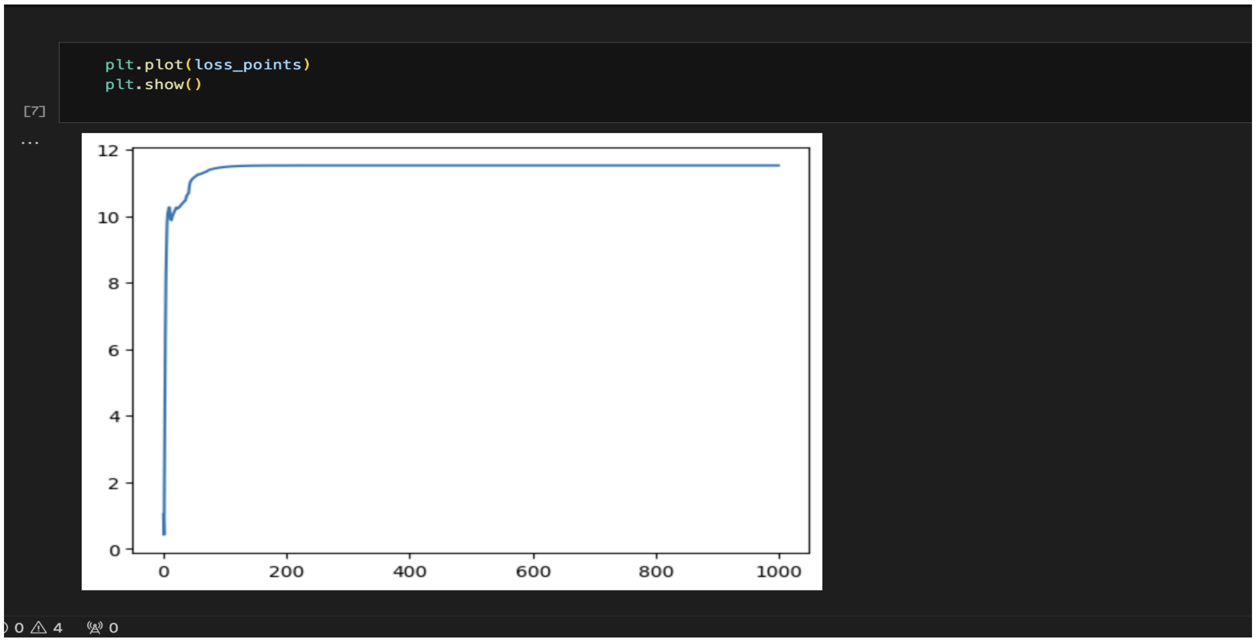
Experiments (I also added the files on git)

1- By changing the Epoch 500, I got very good results While experimenting with different values.



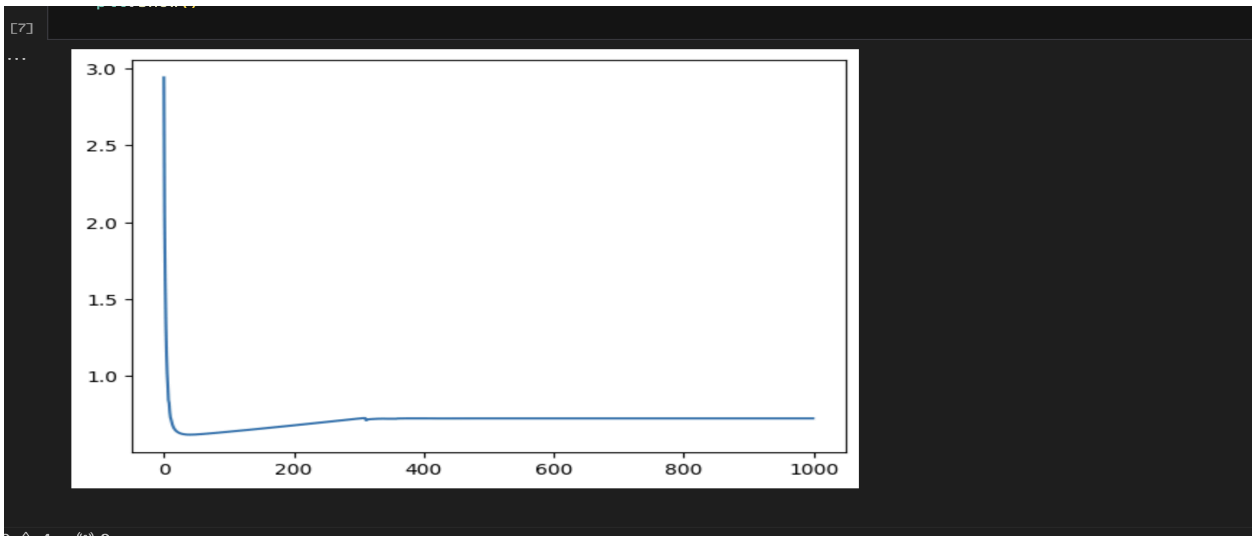
```
plt.plot(loss_points)
plt.show()
```

2- By changing the range and also the numbers of Epoch, I tried different ranges, but with 200, I got the greatest one.



```
plt.plot(loss_points)
plt.show()
```

3- In another experiment, I changed the Learning Rate to different values which is 0.001, and then I also got greate results.



Lastly,

I think tuning is a tricky process, especially when choosing the best parameters like the learning rate (LR), which is crucial along with the dataset's variability and size. TensorFlow is a great solution for utilizing GPUs efficiently during training. For me, it's easier to use TensorFlow, while manual math and coding implementation are difficult.

If we build a neural network manually, it will also be slower. TensorFlow supports large models, whereas manual implementation becomes even more challenging in such cases. The most important thing is debugging—I believe debugging is difficult when working with a complex dataset. However, with TensorFlow, it is a built-in feature, making the process much easier