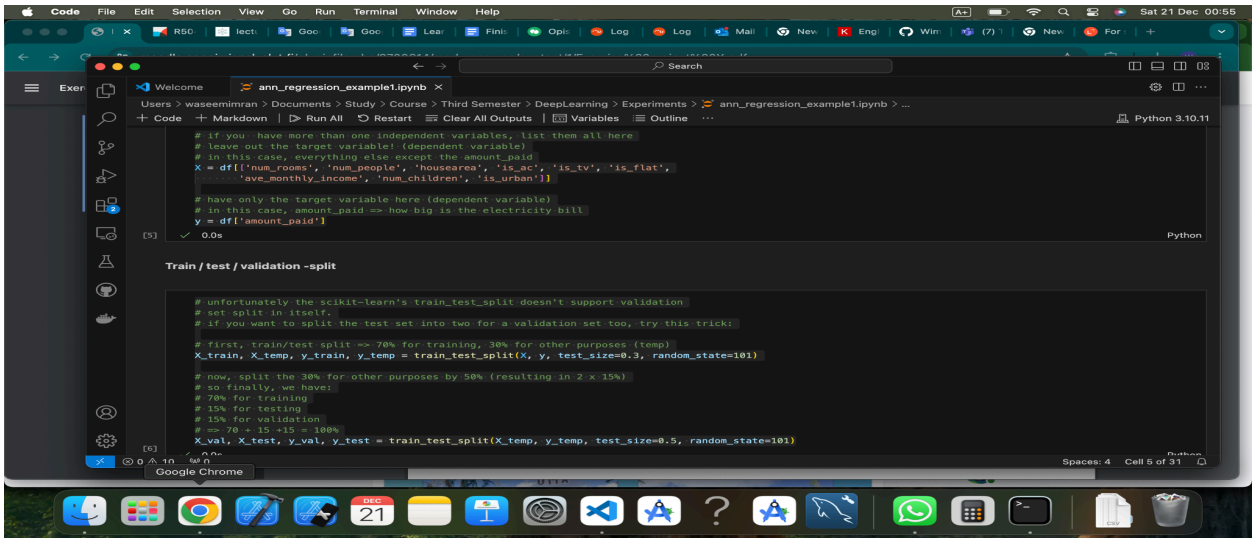
Exercise project X – Experimentation with neural network mathematics

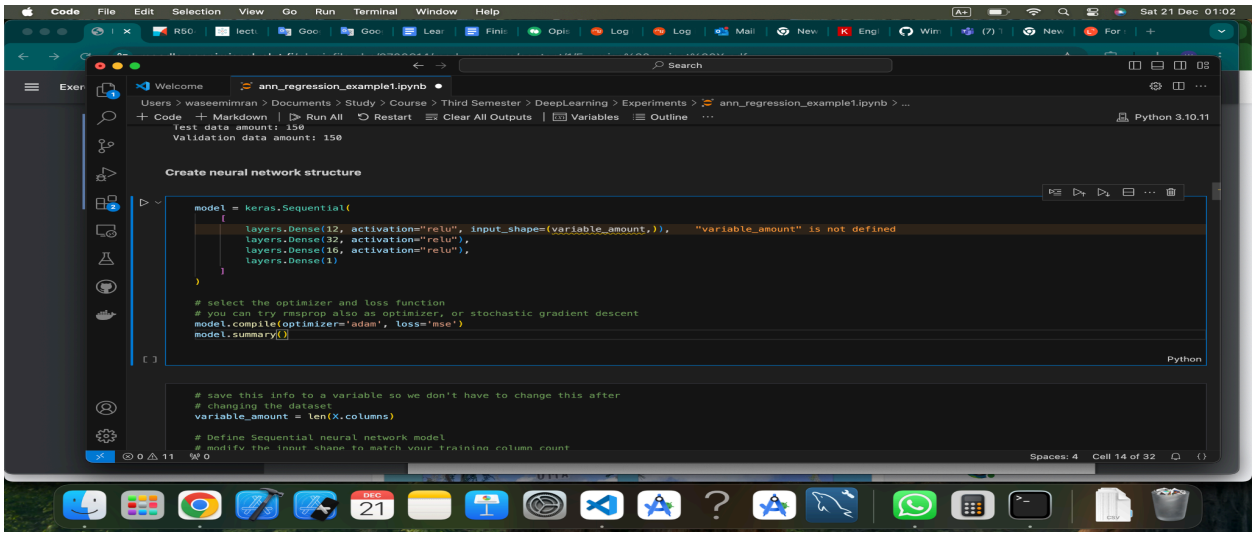
Based on my thoughts, I want to explain how a neural network works.

I am getting the same dataset "Household energy bill data.csv" in my example and also for different experiments.

Data Prepration,
We load the dataset and define the independent and dependent variables. In this case, the independent (X) variable would be num_rooms and num_people etc, but (Y) depends on the variable amount paid.
Data is split into training (70%), validation (15%), and testing (15%) sets to train and evaluate the model.



Neural Network Structure
Input Layer: This takes the number of features as input (in this case, the columns in X).
Hidden Layers: Layers with 12, 32, and 16 nodes, each using the ReLU function to capture non linear relationships.
Output Layer: A node for regression output, since it's predicting a continuous value (amount paid).



Training the Model

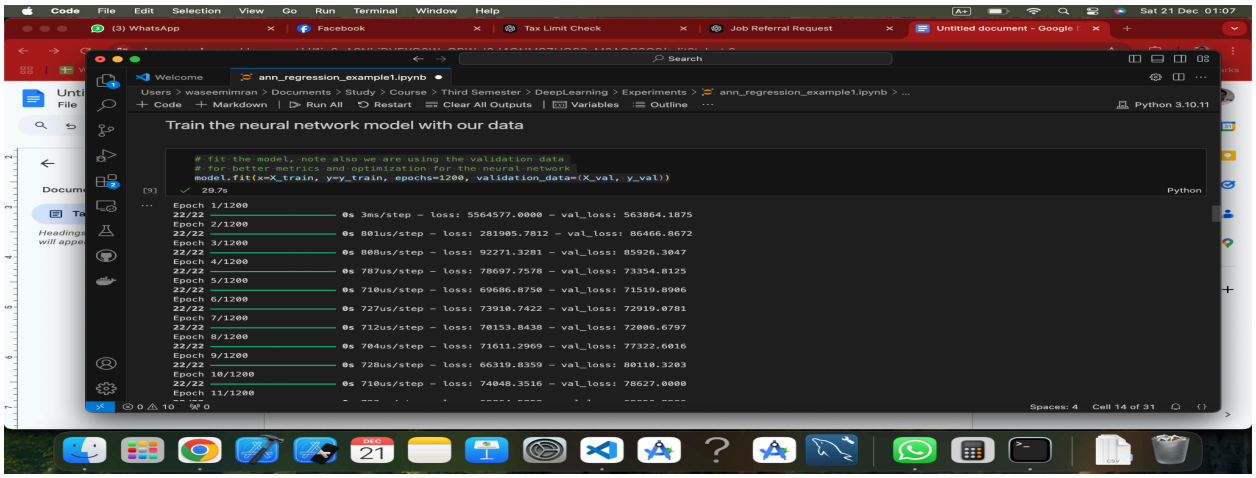
There are three things in training the model.

1- Loss Function: The mse (Mean Squared Error)

2- Optimizer: The Adam optimizer adjusts the weights during training to minimize the loss function.

3- Epochs, model trains numbers of time that we will define.

4- Dropout Layers, are a technique used in neural networks to prevent overfitting.



Evaluation

The model's performance is measured using training and validation data. Metrics like MAE, MSE, RMSE, R-Squared, and Explained Variance Score evaluate

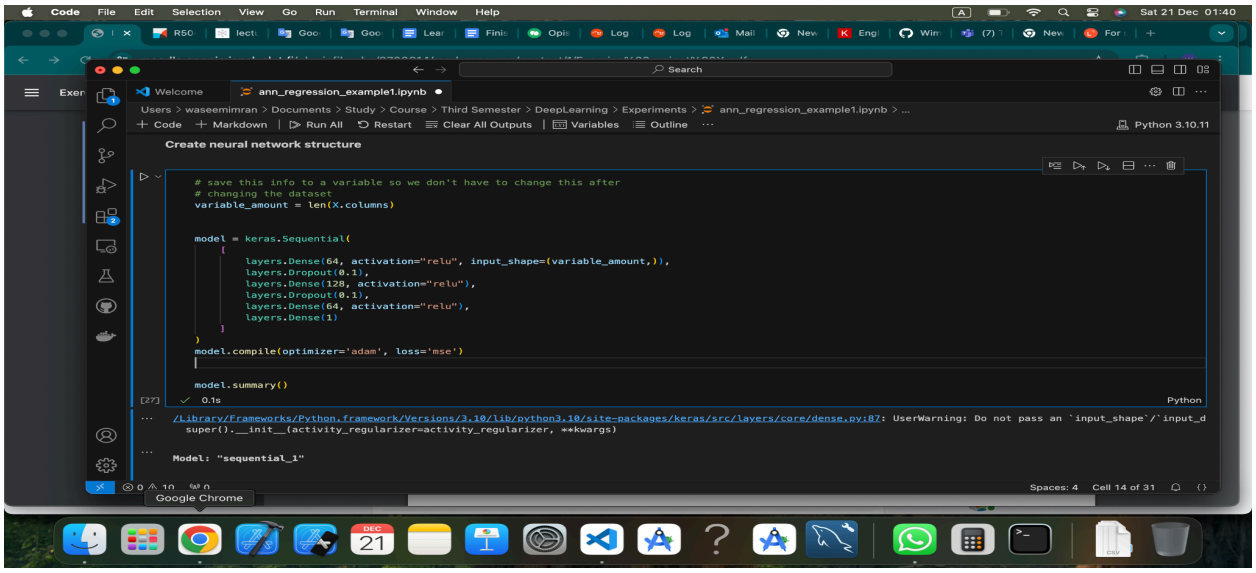
prediction accuracy. Scatter plots and distributions compare predictions to actual values to check for errors.

Last is prediction, after completion of the all steps then we give the information according to the number of variables and predict the results.

(I tried with different data and different changeling like nodes, epochs, dense lyers with different combinations, but now for example I am going to compare here only with last 2 examples)

Let's start different experiments.
First experiments include the changing the layers and epochs amount etc.

1- changing the dense layers along with dropout layers.



There are results.

MAE 73.61 $ MSE 8717.02 $^2 RMSE: 93.36 $ R-squared: 0.73 variance score: 0.83

Befor that, I got the result without dropouts

MAE 64.16 $ MSE 6763.98 $^2 RMSE: 82.24 $ R-squared: 0.79 variance score: 0.82.

1. MAE (Mean Absolute Error):
    - First model: Errors are smaller on average (64.16$), meaning it's more accurate.
    - Second model: Errors are larger (73.61$), so it's less accurate.
2. MSE (Mean Squared Error):
    - First model: Errors are smaller and less sensitive to large mistakes (6763.98$²).
    - Second model: Higher MSE (8717.02$²) suggests the second model makes bigger errors.
3. RMSE (Root Mean Squared Error):
    - First model: Smaller RMSE (82.24$), confirming better accuracy.
    - Second model: Higher RMSE (93.36$), showing it's less accurate.
4. R-squared (R^2):
    - First model: Higher R^2 (0.79), meaning it explains 79% of the variance in the data.
    - Second model: Lower R^2 (0.73), explaining only 73% of the variance.
5. Explained Variance Score:
    - First model: Slightly lower (0.82), but still good.
    - Second model: Marginally better (0.83), meaning it retains variability slightly better.

Conclusion:

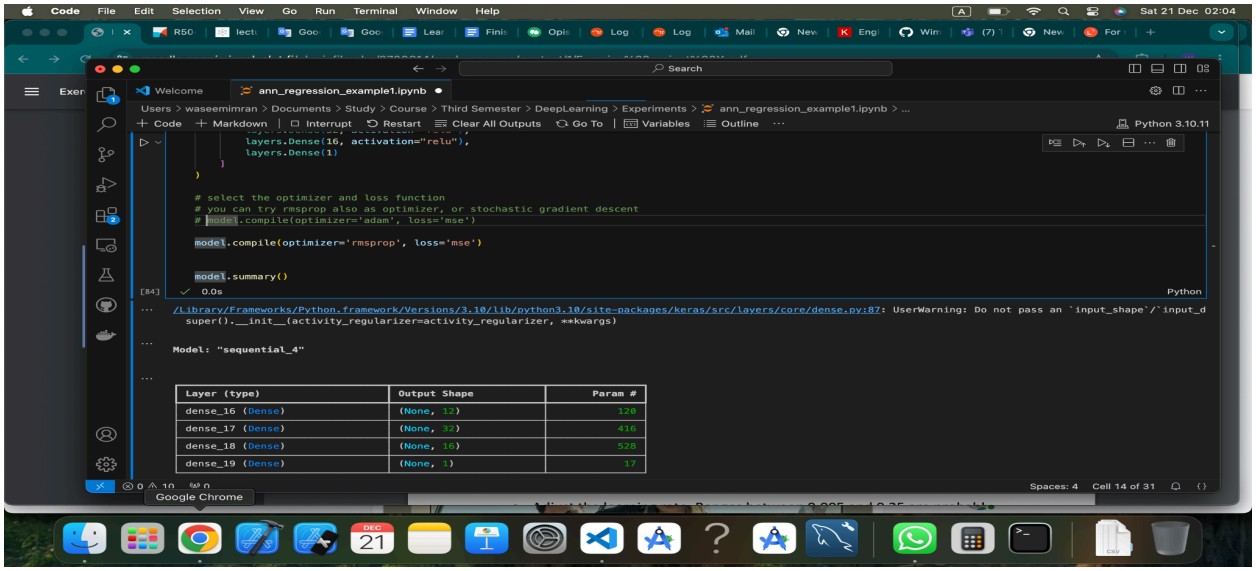
The first model performs better overall, with lower errors and a higher ability to explain the variance. The second model is slightly worse in most metrics except for the explained variance score.

Experiment 2: Using a Different Optimizer

Replace the adam optimizer with rmsprop or sgd to test its effect on convergence and performance.

model.compile(optimizer='rmsprop', loss='mse').

By using this approach, i could not get better result, with Adam i got better result as compared this approach. I got hight MAE and les R-squared with this approach



Other experiments

1- from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)

Its effecting the range of data convereging and might get better results based on data driving.

2- Starting to set a unique value.

I could not get any benefit in setting up a unique randome value at the start of model while setting up custom weight.

3. Learning Rate Adjustment.

Test the effect of learning rate values between 0.005 and 0.25.

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01), loss='mse')

I found it, there is small learning rate might slow but more precise and getting better results.

4. Loss Metrics and Convergence.

Observe the loss plot and model convergence. Mean in what point model is mor stablized.

5. Dataset Size Variation.

In this experiment i got some situation dataset size need to be changed for getting better results, for example as i remember classification, when we have 3 different categories, and 1 of them have totally different amount, then we deduct the size to make it almost similar for getting the better results.

Also, smaller dataset in Neurl network getting heigh MAE and MSE, in my case.

6. TensorFlow vs. Manual Implementation

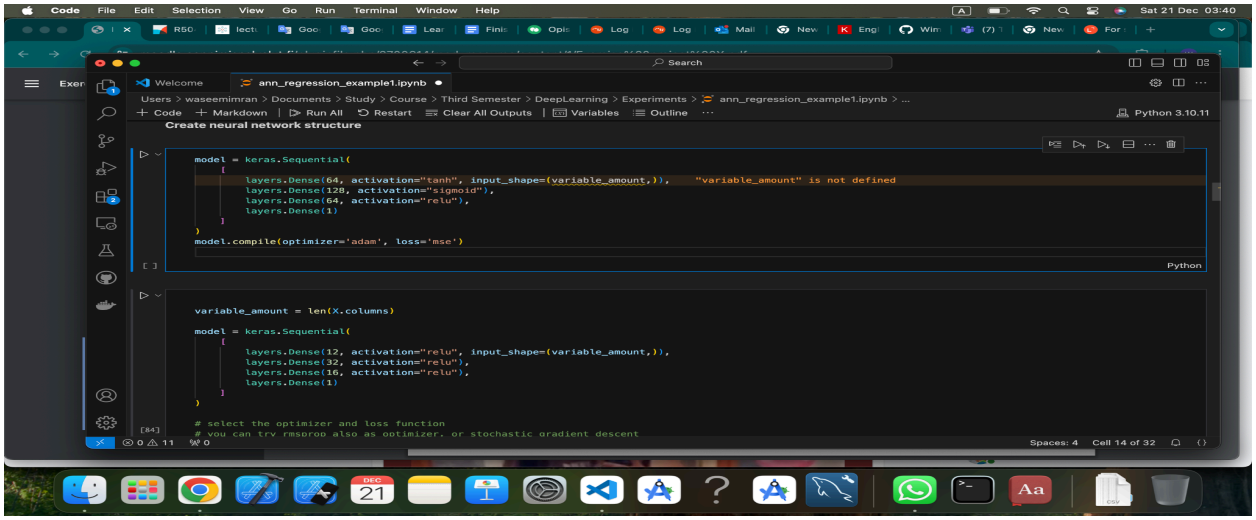
TensorFlow simplifies complex gradient calculations and backpropagation. While Manual implementations allow customization but are prone to errors and inefficiency.

7- Limitations and technical constraints.

Basic networks struggle with large, complex datasets And Sensitive to scaling and input distributions.

8- Experimenting with Different Activation Functions.

There i try to get different positive results with small amount of epochs.



9- Varying the Batch Size

According to my experiments it depends something you need larg batch number to get better schore for example there is a adam and 1200 epoch alsong with 64 batch size, its mean effect the plotting the data in a good way.

model.fit(X_train, y_train, epochs=1200, batch_size=64, validation_data=(X_val, y_val))

10- Applying Early Stopping

In some case early stop perfecting working for overfitting on training the model.

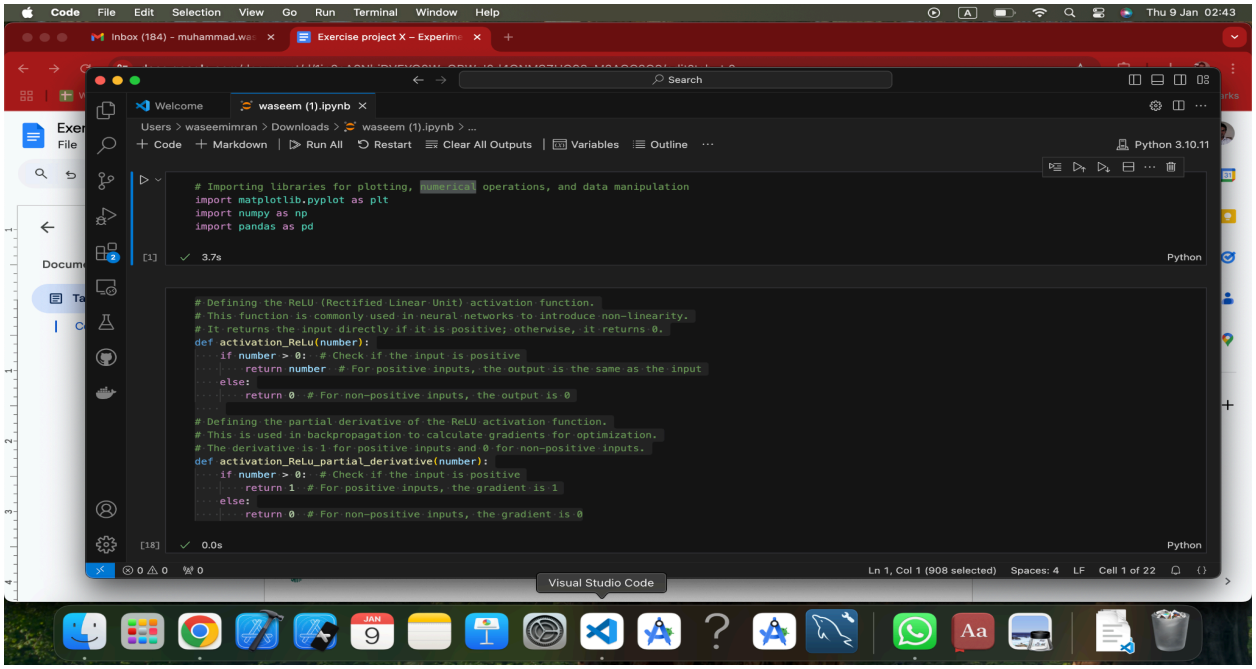
Implement early stopping to halt training once the validation loss stops improving.

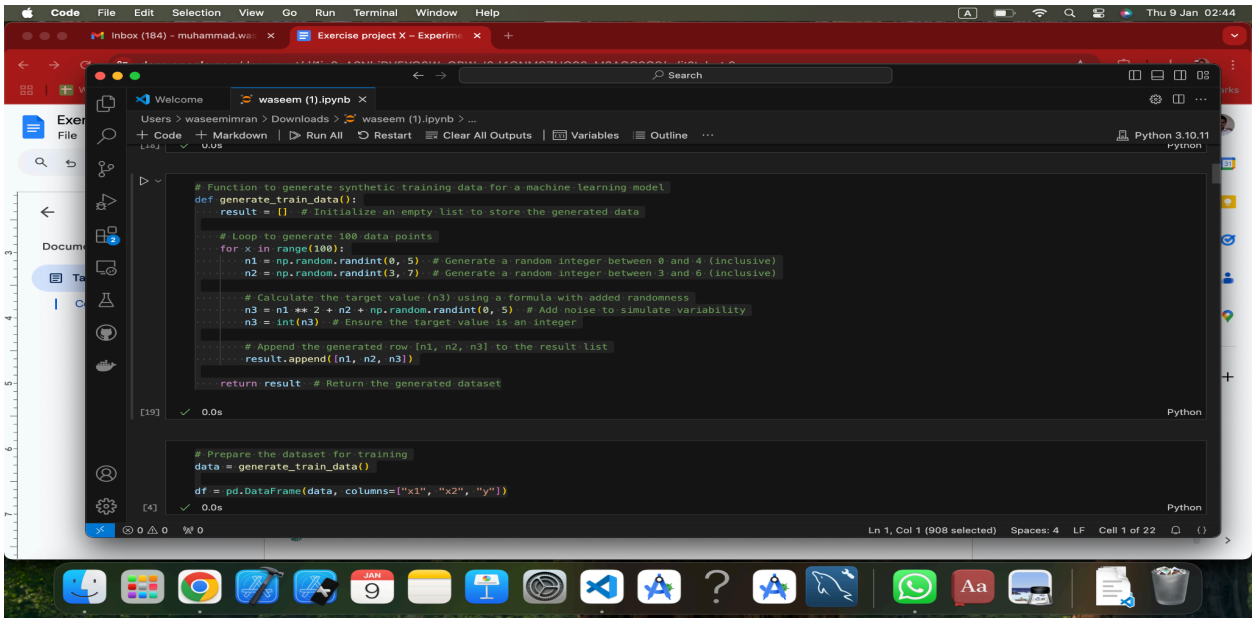
early_stopping = EarlyStopping(monitor='val_loss', patience=50, restore_best_weights=True)

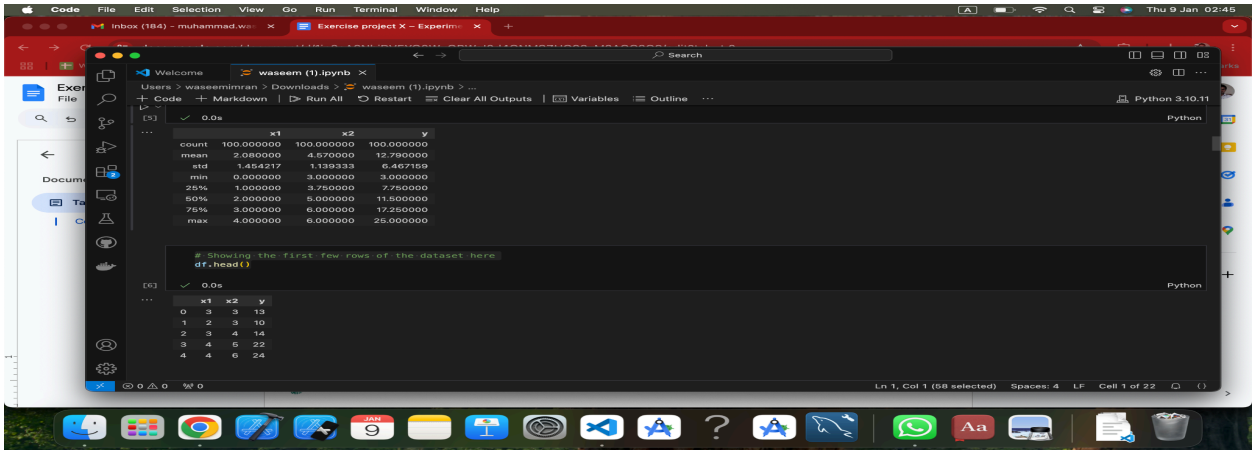
In my case it improve the performance of my model for example i got R squard just 0.6 but with early stop approach i got huge differenecne and got the result around 0.87.

Numerical Operations and Data Manipulation

Here is train a basic neural network step by step. I start with sample data and define the ReLU activation function and its derivative. Then, it initializes weights and biases and uses backpropagation with gradient descent to improve them. The model reduces error over several training rounds (epochs). Finally, it uses the trained weights and biases to make predictions, shows how the error decreases during training, and adjusts the predictions back to the original scale.

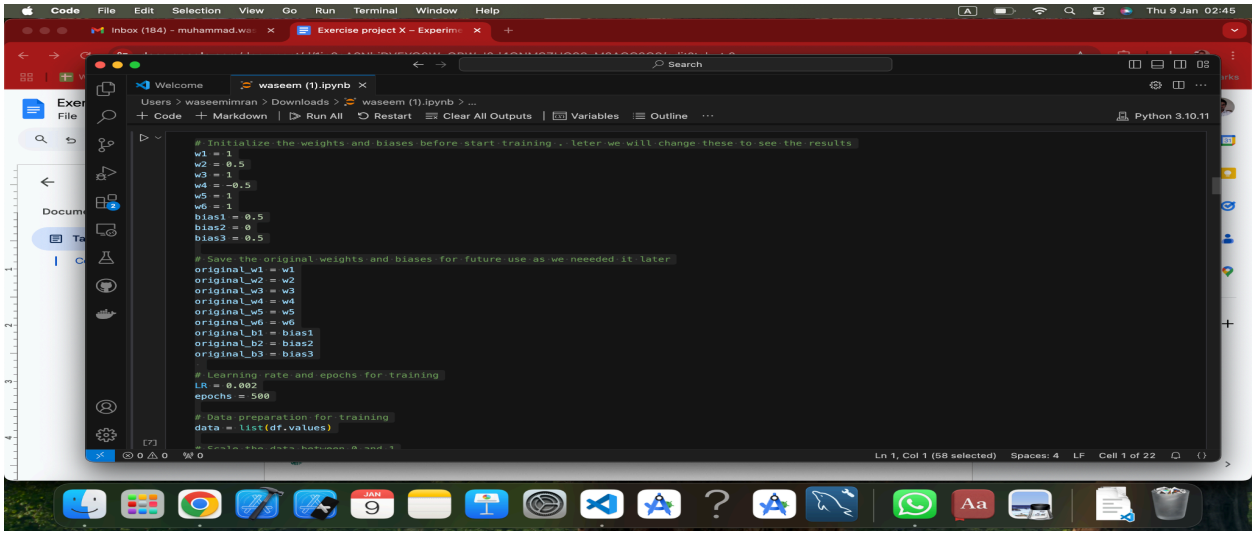




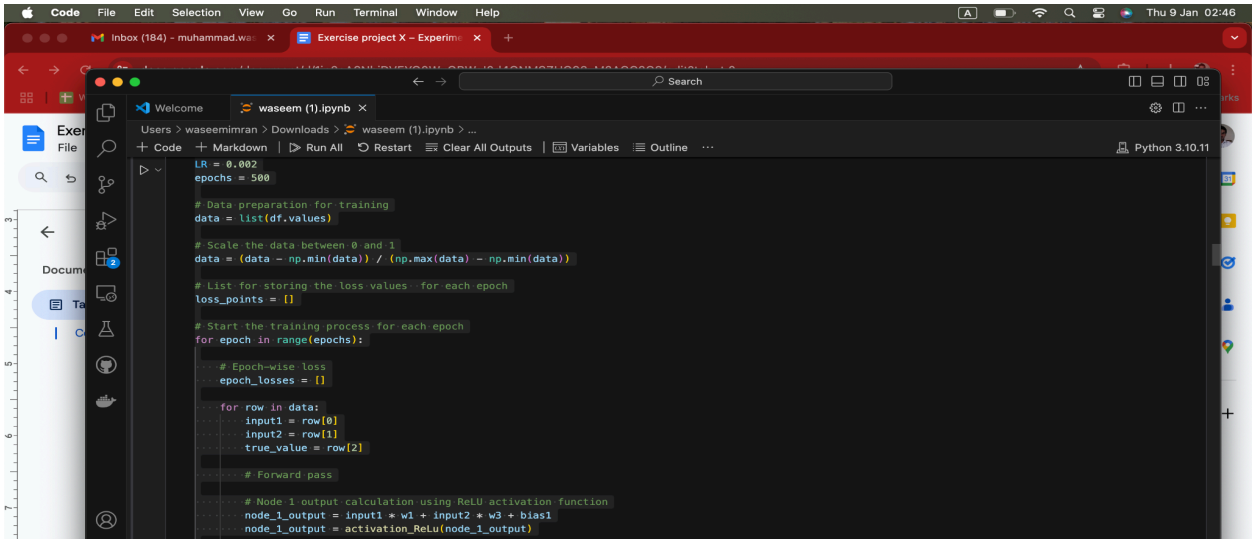


```
# Showing the first few rows of the dataset here
df.head()
```

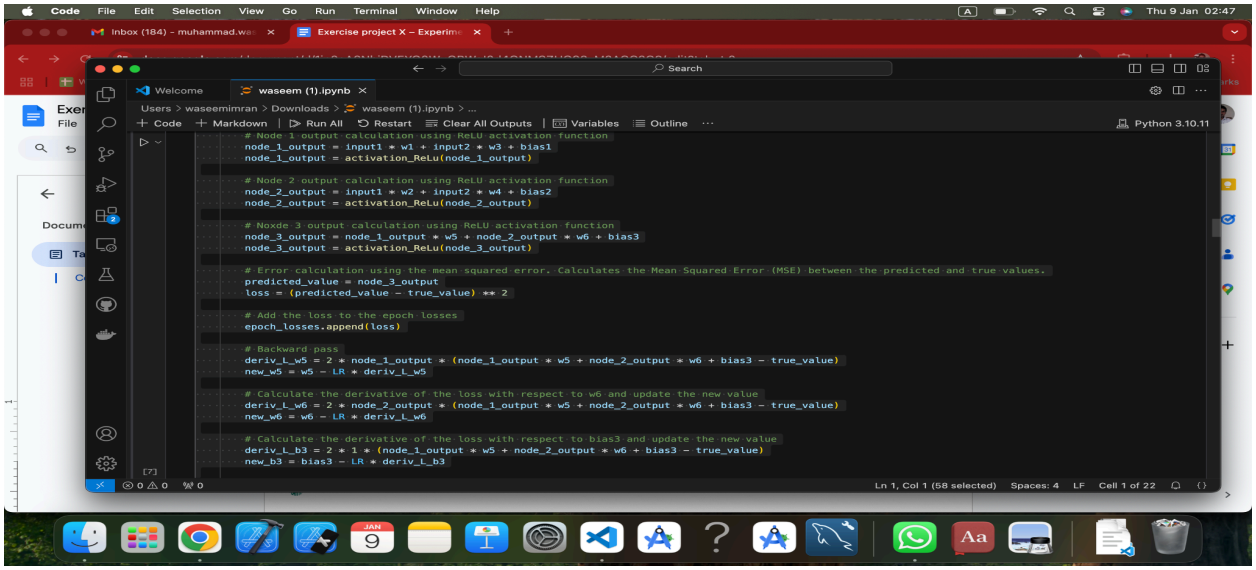
	x1	x2	y
0	3	3	13
1	2	3	10
2	3	4	14
3	4	5	22
4	4	6	24



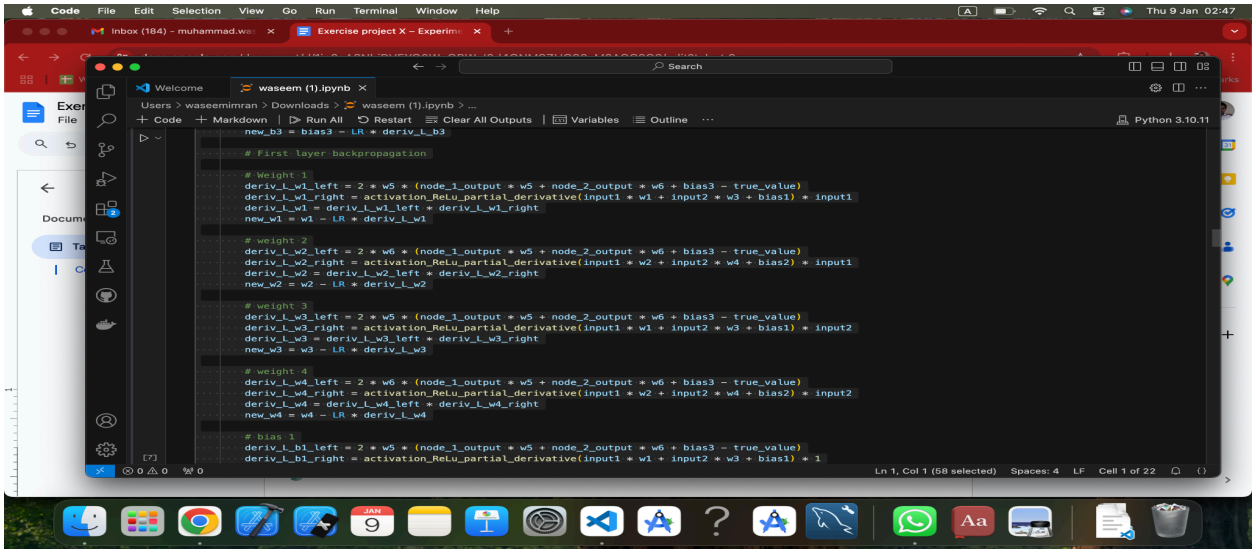
```
# Initialize the weights and biases before start training , leter we will change these to see the results
w1 = 1
w2 = 0.5
w3 = 1
w4 = -0.5
w5 = 1
w6 = 1
bias1 = 0.5
bias2 = 0
bias3 = 0.5

# Save the original weights and biases for future use as we neeeded it later
original_w1 = w1
original_w2 = w2
original_w3 = w3
original_w4 = w4
original_w5 = w5
original_w6 = w6
original_b1 = bias1
original_b2 = bias2
original_b3 = bias3

# Learning rate and epochs for training
LR = 0.002
epochs = 500

# Data preparation for training
data = list(df.values)
```



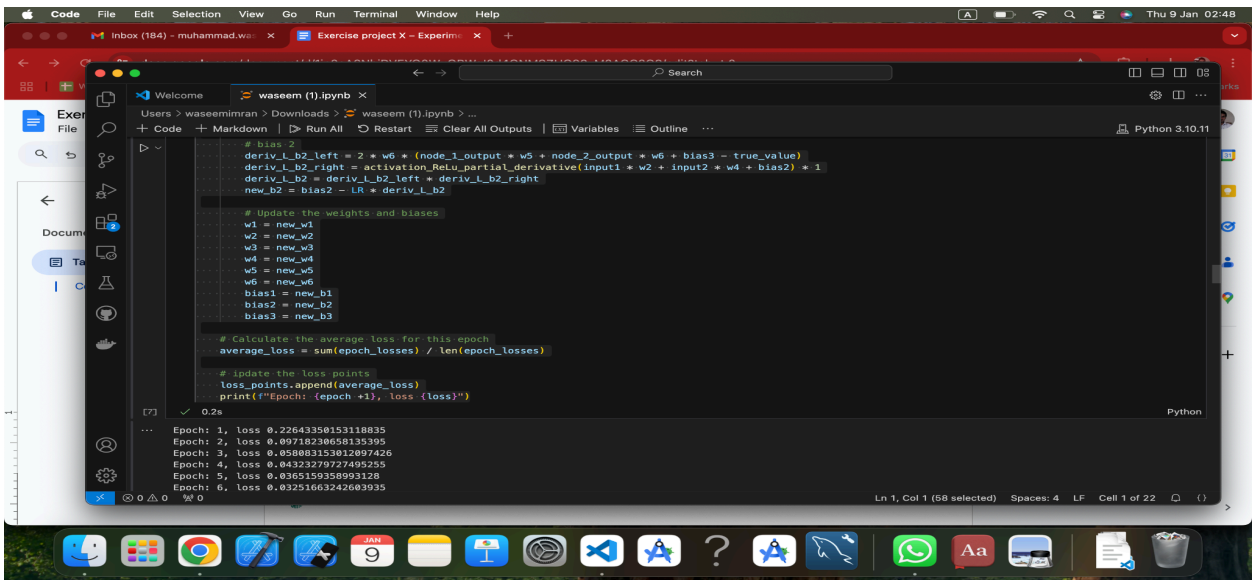
```
LR = 0.002
epochs = 500

# Data preparation for training
data = list(df.values)

# Scale the data between 0 and 1
data = (data - np.min(data)) / (np.max(data) - np.min(data))

# List for storing the loss values for each epoch
loss_points = []

# Start the training process for each epoch
for epoch in range(epochs):

    # Epoch-wise loss
    epoch_losses = []

    for row in data:
        input1 = row[0]
        input2 = row[1]
        true_value = row[2]

        # Forward pass

        # Node 1 output calculation using ReLU activation function
        node_1_output = input1 * w1 + input2 * w3 + bias1
        node_1_output = activation_ReLu(node_1_output)

        # Node 2 output calculation using ReLU activation function
        node 2 output = input1 * w2 + input2 * w4 + bias2
```



```
        # Node 1 output calculation using ReLU activation function
        node_1_output = input1 * w1 + input2 * w3 + bias1
        node_1_output = activation_ReLu(node_1_output)

        # Node 2 output calculation using ReLU activation function
        node_2_output = input1 * w2 + input2 * w4 + bias2
        node_2_output = activation_ReLu(node_2_output)

        # Noxde 3 output calculation using ReLU activation function
        node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
        node_3_output = activation_ReLu(node_3_output)

        # Error calculation using the mean squared error. Calculates the Mean Squared Error (MSE) between the predicted and true values.
        predicted_value = node_3_output
        loss = (predicted_value - true_value) ** 2

        # Add the loss to the epoch losses
        epoch_losses.append(loss)

        # Backward pass
        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

        # Calculate the derivative of the loss with respect to 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

        # Calculate the derivative of the loss with respect to 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
```



```
        new_b3 = bias3 - LR * deriv_L_b3

        # First layer backpropagation

        # Weight 1
        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

        # 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

        # 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

        # 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

        # 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
```



```
        # 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

        # Update the weights and biases
        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

    # Calculate the average loss for this epoch
    average_loss = sum(epoch_losses) / len(epoch_losses)

    # ipdate the loss points
    loss_points.append(average_loss)
    print(f"Epoch: {epoch +1}, loss {loss}")
```

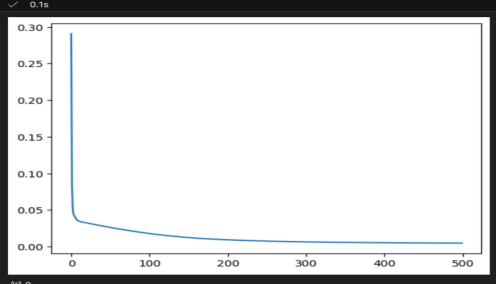
0.2s

Python

```
Epoch: 1, loss 0.22643350153118835
Epoch: 2, loss 0.09718230658135395
Epoch: 3, loss 0.05808315301209742 6
Epoch: 4, loss 0.043232797274952 55
Epoch: 5, loss 0.0365159358993128
Epoch: 6, loss 0.03251663242603935
```

```
# Print the original and new weights and biases
print("ORIGINAL WEIGHTS AND BIASES")
print(f"w1: {original_w1}")
print(f"w2: {original_w2}")
print(f"w3: {original_w3}")
print(f"w4: {original_w4}")
print(f"w5: {original_w5}")
print(f"w6: {original_w6}")
print(f"b1: {original_b1}")
print(f"b2: {original_b2}")
print(f"b3: {original_b3}")

print("\n\n####################################")

print("NEW WEIGHTS AND BIASES")
print(f"w1: {w1}")
print(f"w2: {w2}")
print(f"w3: {w3}")
print(f"w4: {w4}")
print(f"w5: {w5}")
print(f"w6: {w6}")
print(f"b1: {bias1}")
print(f"b2: {bias2}")
print(f"b3: {bias3}")
```

```
[8]  0.0s                                                    Python
...  ORIGINAL WEIGHTS AND BIASES
```




```
# Function to predict the output
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
    node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
    node_3_output = activation_ReLu(node_3_output)

    return node_3_output
```

```
[10]  0.0s                                                   Python
```

```
# Random test
df.iloc[5]
```

```
[11]  0.0s                                                   Python
...  x1    3
     x2    3
     y    16
```

```
# Check the prediction data
data[5]
```

```
array([0.12, 0.12, 0.64])
```

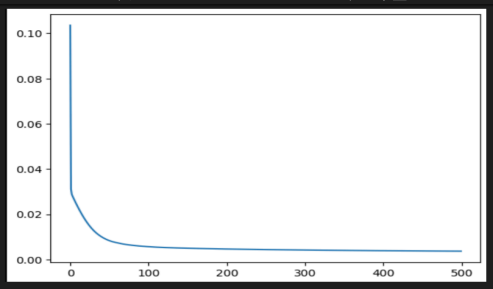
```
# Predict the output while passing the input values
result = predict(0.03846154, 0.15384615)
result
```

```
0.27124116670118975
```

```
# Finally scale the result back to the original scale
df['y'].max() * result
```

```
6.781029167529744
```

```
# Experiment with different values
# change data range to 500. Then we have very good results for prediction and for loss. we have improved the model
```




```
x1    3
x2    5
y    16
Name: 5, dtype: int64
```

```
x1    3
x2    5
y    16
Name: 5, dtype: int64
```

```
# Check the prediction data
data[5]
```

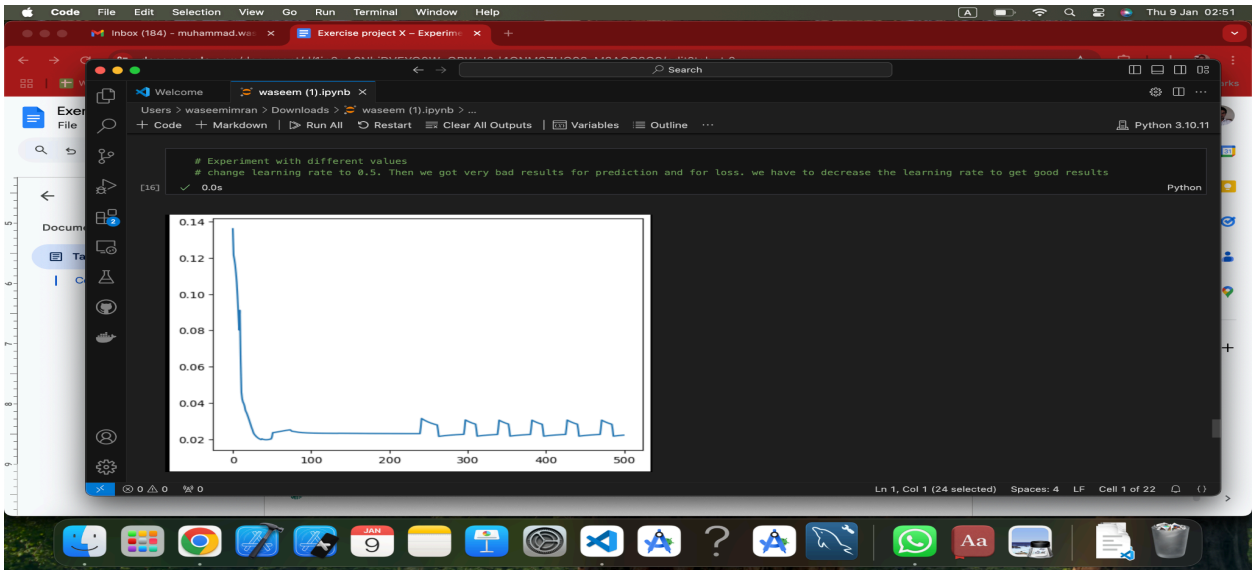
```
array([0.11538462, 0.19230769, 0.61538462])
```

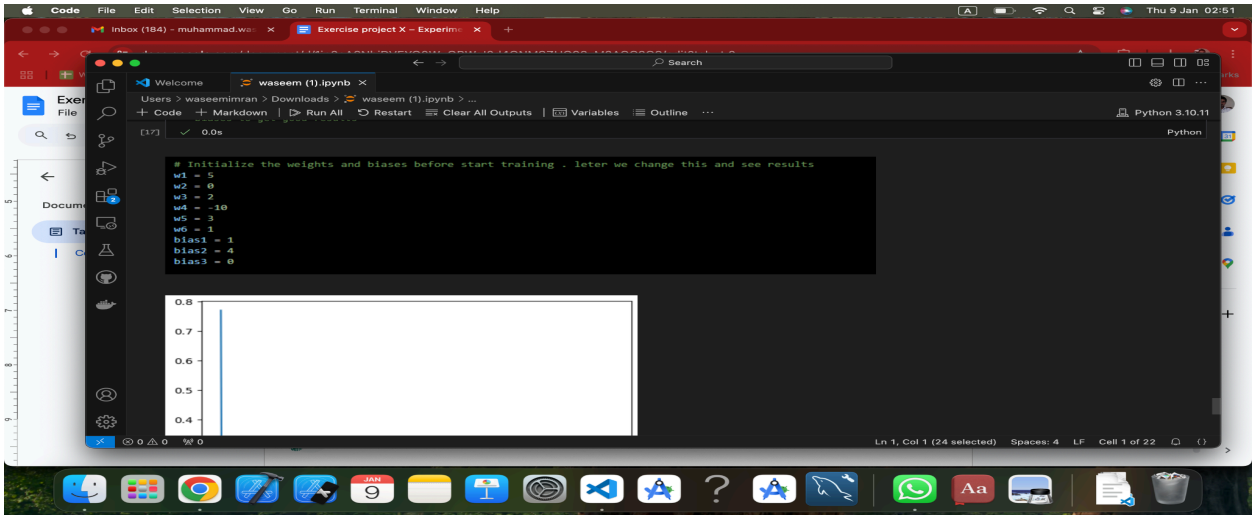
```
# Predict the output while passing the input values
result = predict(0.11538462, 0.19230769)
result
```

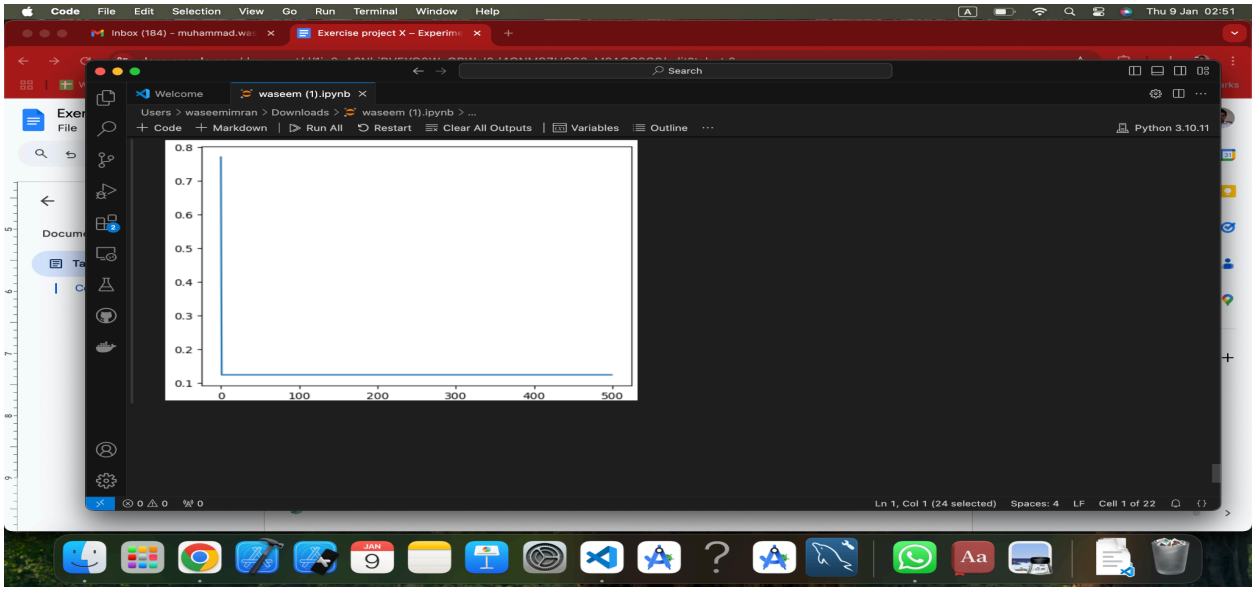
```
0.6355243808743919
```

```
# Finally scale the result back to the original scale
df['y'].max() * result
```

```
16.52363390273419
```


Welcome
waseem (1).ipynb
Users > waseemimran > Downloads > waseem (1).ipynb > ...
Code
Markdown
Run All
Restart
Clear All Outputs
Variables
Outline
Python 3.10.11
Experiment with different values
change learning rate to 0.5. Then we got very bad results for prediction and for loss. we have to decrease the learning rate to get good results
[16] 0.0s
Python
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
100
200
300
400
500
Ln 1, Col 1 (24 selected)
Spaces: 4
LF
Cell 1 of 22


Welcome
waseem (1).ipynb
Users > waseemimran > Downloads > waseem (1).ipynb > ...
Code
Markdown
Run All
Restart
Clear All Outputs
Variables
Outline
Python 3.10.11
[17] 0.0s
Python
# Initialize the weights and biases before start training . leter we change this and see results
w1 = 5
w2 = 0
w3 = 2
w4 = -10
w5 = 3
w6 = 1
bias1 = 1
bias2 = 4
bias3 = 0
0.8
0.7
0.6
0.5
0.4
Ln 1, Col 1 (24 selected)
Spaces: 4
LF
Cell 1 of 22


Welcome
waseem (1).ipynb
Users > waseemimran > Downloads > waseem (1).ipynb > ...
Code
Markdown
Run All
Restart
Clear All Outputs
Variables
Outline
Python 3.10.11
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
Ln 1, Col 1 (24 selected)
Spaces: 4
LF
Cell 1 of 22