

## Exercise project X – Experimentation with neural network mathematics

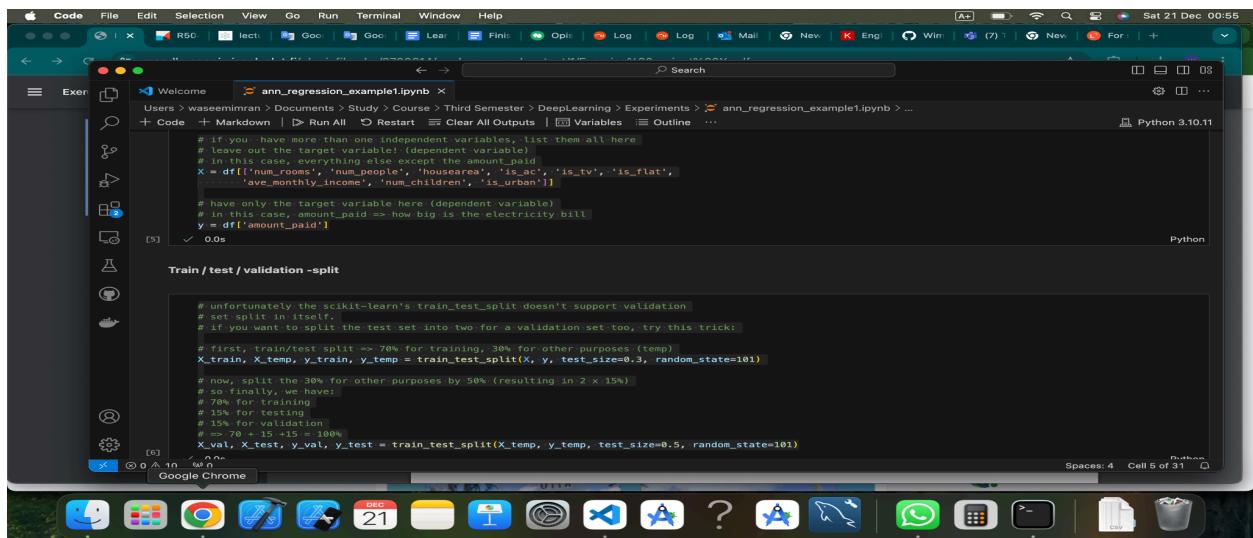
Based on my thoughts, I want to explain how a neural network works.

I am getting the same dataset “[Household energy bill data.csv](#)” in my example and also for different experiments.

Data Preparation,

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.



The screenshot shows a Jupyter Notebook interface with a Python 3.10.11 kernel. The code cell contains the following:

```
# if you have more than one independent variables, list them all here
# leave out the target variable! (dependent variable)
# in this case, everything else except the amount_paid
X = df[['num_rooms', 'num_people', 'numareas', 'is_ac', 'is_tv', 'is_flat',
        'ave_monthly_income', 'num_children', 'is_urban']]
# have only the target variable here (dependent variable)
# in this case, amount_paid => how big is the electricity bill
y = df['amount_paid']

0.0s
```

Below the code cell, the notebook displays the following text:

Train / test / validation -split

```
# unfortunately the scikit-learn's train_test_split doesn't support validation
# set split in itself.
# if you want to split the test set into two for a validation set too, try this trick:
# first, train/test split => 70% for training, 30% for other purposes (temp)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=101)

# now, split the 30% for other purposes by 50% (resulting in 2 x 15%)
# so finally, we have:
# 70% for training
# 15% for testing
# 15% for validation
# == 70 + 15 +15 = 100%
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=101)
```

## 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).

```

Code File Edit Selection View Go Run Terminal Window Help
Users > waseemimran > Documents > Study > Course > Third Semester > DeepLearning > Experiments > ann_regression_example1.ipynb ...
+ Code + Markdown | ▶ Run All ⌘ Restart ⌘ Clear All Outputs Variables Outline ...
Sat 21 Dec 01:02 Python 3.10.11

Create neural network structure

model = keras.Sequential(
    [
        layers.Dense(12, activation='relu', input_shape=(variable_amount)), "variable_amount" is not defined
        layers.Dense(12, activation='relu'),
        layers.Dense(16, activation='relu'),
        layers.Dense(1)
    ]
)

# select the optimizer and loss function
# you can try rmsprop also as optimizer, or stochastic gradient descent
model.compile(optimizer='adam', loss='mse')
model.summary()

# save this info to a Variable so we don't have to change this after
# changing the dataset
variable_amount = len(X.columns)

# Define Sequential neural network model
# modify the input shape to match your training column count

```

Spaces: 4 Cell 14 of 32

## 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.

```

Train the neural network model with our data

# fit the model, note also we are using the validation data
# for better metrics and optimization for the neural network
model.fit(x=X_train, y=y_train, epochs=1200, validation_data=(X_val, y_val))

29.7s
Epoch 1/1200
22/22 - 0s 3ms/step - loss: 5564577.0000 - val_loss: 563864.1875
Epoch 2/1200
22/22 - 0s 80us/step - loss: 281985.7812 - val_loss: 86466.8672
Epoch 3/1200
22/22 - 0s 80us/step - loss: 92271.3781 - val_loss: 85926.3047
Epoch 4/1200
22/22 - 0s 787us/step - loss: 78697.7578 - val_loss: 73354.8125
Epoch 5/1200
22/22 - 0s 710us/step - loss: 69686.8758 - val_loss: 71519.8986
Epoch 6/1200
22/22 - 0s 727us/step - loss: 73910.7422 - val_loss: 72919.0781
Epoch 7/1200
22/22 - 0s 732us/step - loss: 70153.8438 - val_loss: 72086.6797
Epoch 8/1200
22/22 - 0s 784us/step - loss: 71611.2969 - val_loss: 77322.6816
Epoch 9/1200
22/22 - 0s 728us/step - loss: 66319.8559 - val_loss: 80118.3203
Epoch 10/1200
22/22 - 0s 710us/step - loss: 74648.3516 - val_loss: 78627.0000
Epoch 11/1200
22/22 - 0s 710us/step - loss: 74648.3516 - val_loss: 78627.0000

```

Spaces: 4 Cell 14 of 31

## 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.

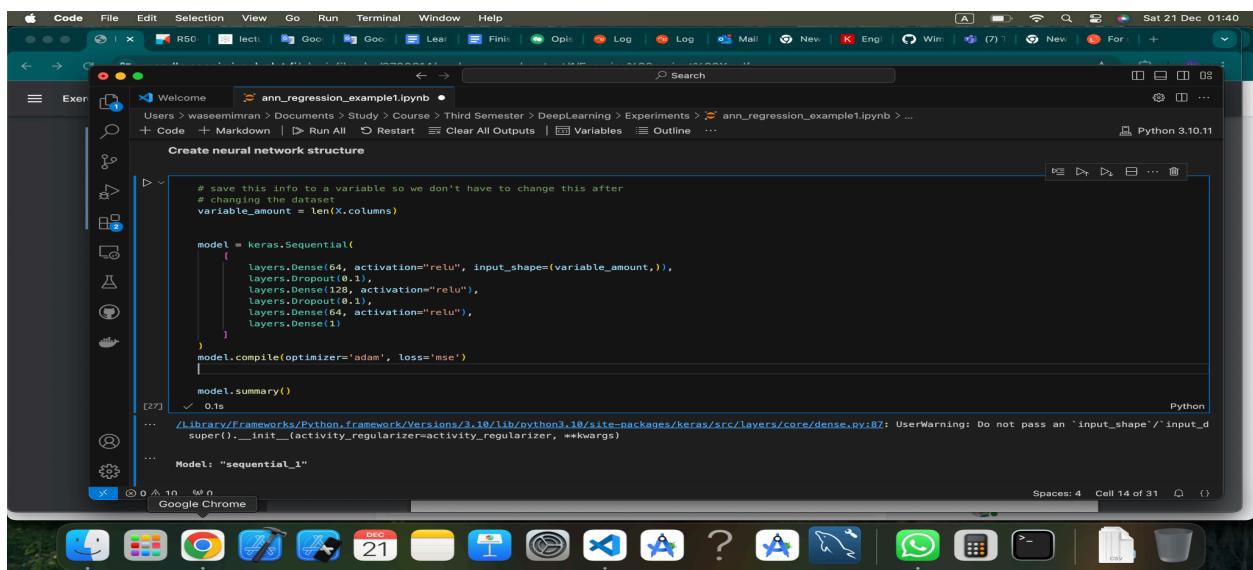
Last is prediction, after completion of the all steps then we give the information according to the number of variables and predict the results.

**(I tried with different data and different changeling like nodes, epochs, dense layers 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.



The screenshot shows a Jupyter Notebook interface on a Mac OS X desktop. The window title is 'Welcome' and the file path is 'Users > waseemimran > Documents > Study > Course > Third Semester > DeepLearning > Experiments > ann\_regression\_example1.ipynb'. The notebook contains the following Python code:

```
# save this info to a variable so we don't have to change this after
# changing the dataset
variable_amount = len(X.columns)

model = keras.Sequential(
    [
        layers.Dense(64, activation="relu", input_shape=(variable_amount,)),
        layers.Dropout(0.1),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.1),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ]
)
model.compile(optimizer='adam', loss='mse')

model.summary()

```

The output of the code shows the model summary:

```
Model: "sequential_1"
Python
0.1s
... /Library/Frameworks/Python_Framework/Versions/3.10/lib/python3.10/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape` / `input_d
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
...
Model: "sequential_1"
```

The status bar at the bottom right indicates 'Spaces: 4 Cell 14 of 31'.

There are results.

MAE 73.61 \$ MSE 8717.02 \$<sup>2</sup> 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 \$<sup>2</sup> 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\$<sup>2</sup>).
- Second model: Higher MSE (8717.02\$<sup>2</sup>) 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

The screenshot shows a Jupyter Notebook interface on a Mac OS X desktop. The notebook is titled 'ann\_regression\_example1.ipynb'. The code cell contains the following Python code:

```
layers.Dense(16, activation="relu"),
layers.Dense(1)

# select the optimizer and loss function
# you can try rmsprop also as optimizer, or stochastic gradient descent
# model.compile(optimizer='adam', loss='mse')

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

model.summary()
```

The output cell shows the model summary:

```
Model: "sequential_4"

Layer (type)          Output Shape       Param #
dense_16 (Dense)     (None, 12)           120
dense_17 (Dense)     (None, 32)           416
dense_18 (Dense)     (None, 16)           528
dense_19 (Dense)     (None, 1)            17

Total params: 1,063
Trainable params: 1,063
Non-trainable params: 0
Weights: 531
```

The status bar at the bottom indicates 'Spaces: 4 Cell 14 of 31'.

## Other experiments

```
1- from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)
```

Its effecting the range of data converging 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 more stabilized.

### 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.

The screenshot shows a Jupyter Notebook interface on a Mac OS X desktop. The top menu bar includes Code, File, Edit, Selection, View, Go, Run, Terminal, Window, Help, and a search bar. The status bar at the bottom right shows the date as Sat 21 Dec 03:40 and Python version 3.10.11. The main notebook area displays two code cells. The first cell contains:`model = keras.Sequential([
 layers.Dense(64, activation="tanh", input_shape=(variable_amount)), "variable_amount" is not defined
 layers.Dense(64, activation="sigmoid"),
 layers.Dense(64, activation="relu"),
 layers.Dense(1)
])
model.compile(optimizer='adam', loss='mse')`

The second cell contains:`variable_amount = len(X.columns)

model = keras.Sequential([
 layers.Dense(12, activation="relu", input_shape=(variable_amount)),
 layers.Dense(32, activation="relu"),
 layers.Dense(16, activation="relu"),
 layers.Dense(1)
])

# select the optimizer and loss function
# you can try rmsprop also as optimizer. or stochastic gradient descent`

## 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.

```
# Importing libraries for plotting, numerical operations, and data manipulation
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Defining the ReLU (Rectified Linear Unit) activation function.
# This function is commonly used in neural networks to introduce non-linearity.
# It returns the input directly if it is positive; otherwise, it returns 0.
def activation_ReLU(number):
    if number > 0: # Check if the input is positive
        return number # For positive inputs, the output is the same as the input
    else:
        return 0 # For non-positive inputs, the output is 0

# Defining the partial derivative of the ReLU activation function.
# This is used in backpropagation to calculate gradients for optimization.
# The derivative is 1 for positive inputs and 0 for non-positive inputs.
def activation_ReLU_partial_derivative(number):
    if number > 0: # Check if the input is positive
        return 1 # For positive inputs, the gradient is 1
    else:
        return 0 # For non-positive inputs, the gradient is 0
```

```
# Function to generate synthetic training data for a machine learning model
def generate_train_data():
    result = [] # Initialize an empty list to store the generated data

    # Loop to generate 100 data points
    for x in range(100):
        n1 = np.random.randint(0, 5) # Generate a random integer between 0 and 4 (inclusive)
        n2 = np.random.randint(3, 7) # Generate a random integer between 3 and 6 (inclusive)

        # Calculate the target value (n3) using a formula with added randomness
        n3 = n1 ** 2 + n2 + np.random.randint(0, 5) # Add noise to simulate variability
        n3 = int(n3) # Ensure the target value is an integer

        # Append the generated row [n1, n2, n3] to the result list
        result.append([n1, n2, n3])

    return result # Return the generated dataset

# Prepare the dataset for training
data = generate_train_data()

df = pd.DataFrame(data, columns=["x1", "x2", "y"])
✓ 0.0s
```

Code Editor:

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

Output:

```
0.0s  
...  
count    100.000000  100.000000  100.000000  
mean     2.080000   4.570000   12.790000  
std      1.454217   1.139333   6.467159  
min     -0.500000   0.000000   3.000000  
25%    1.000000   3.750000   7.750000  
50%    2.000000   6.000000  11.500000  
75%    3.000000   6.000000  17.250000  
max     4.000000   6.000000  25.000000
```

Code Editor:

```
# Initialize the weights and biases before start training . later 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 needed it later  
original_w1 = w1  
original_w2 = w2  
original_w3 = w3  
original_w4 = w4  
original_w5 = w5  
original_w6 = w6  
original_bias1 = bias1  
original_bias2 = bias2  
original_bias3 = bias3  
  
# Learning rate and epochs for training  
LR = 0.002  
epochs = 500  
  
# Data preparation for training  
data = list(df.values)
```

Code Editor:

```
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 * w2 + 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)

# Node 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_w1 = 2 * node_1_output * (node_1_output * w5 + node_2_output * w6 + bias3 - true_value)
new_w1 = w1 - LR * deriv_L_w1

# 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
```

```
# 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 * w2 + 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_3_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
deriv_L_b1 = deriv_L_b1_left * deriv_L_b1_right
new_b1 = b1 - LR * deriv_L_b1
```

```
# 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 = b2 - 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)

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

... Epoch: 1, loss: 0.22643258153118835
Epoch: 2, loss: 0.09718239650135399
Epoch: 3, loss: 0.058083153012097426
Epoch: 4, loss: 0.04323279727495255
Epoch: 5, loss: 0.0365159358993128
Epoch: 6, loss: 0.03251663242603935
```

Code File Edit Selection View Go Run Terminal Window Help

Thu 9 Jan 02:49

Inbox (184) - muhammad.was ... Exercise project X - Experiment ... +

Welcome waseem (1).ipynb

Users > waseemirman > Downloads > waseem (1).ipynb > ...

+ Code + Markdown | ▶ Run All ⌘ Restart ⌘ Clear All Outputs Variables Outline ... Python 3.10.11

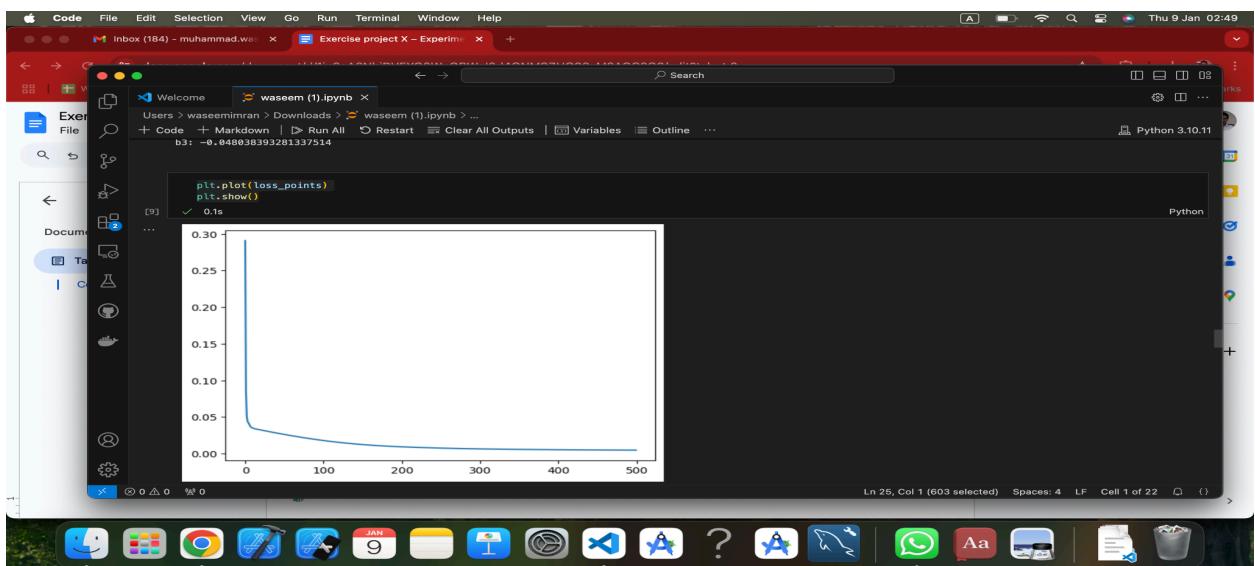
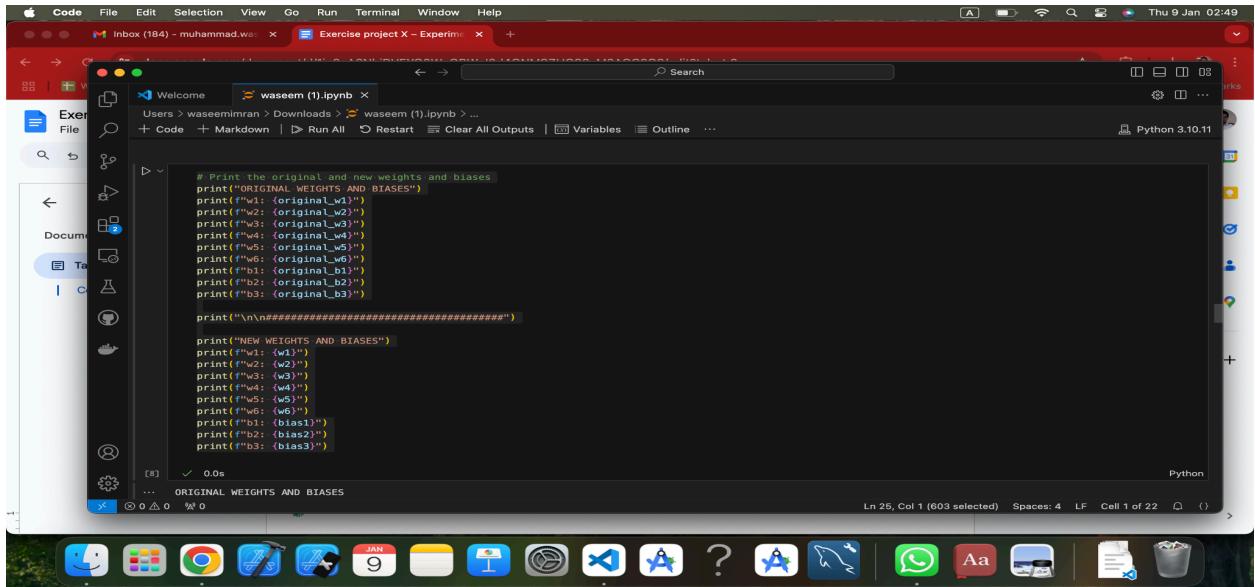
```
# Print the original and new weights and biases
print("ORIGINAL WEIGHTS AND BIASES")
print(f"w1: {original_w1}")
print(f"w2: {original_w2}")
print(f"w3: {original_w3}")
print(f"w4: {original_w4}")
print(f"w5: {original_w5}")
print(f"w6: {original_w6}")
print(f"b1: {original_b1}")
print(f"b2: {original_b2}")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

print("\n\nNEW 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}")
```

0.0s

ORIGINAL WEIGHTS AND BIASES

Ln 25, Col 1 (603 selected) Spaces: 4 LF Cell 1 of 22



Code File Edit Selection View Go Run Terminal Window Help

Thu 9 Jan 02:49

Inbox (184) - muhammad.was ... Exercise project X - Experiment ... +

Welcome waseem (1).ipynb

Users > waseemirman > Downloads > waseem (1).ipynb > ...

+ Code + Markdown | ▶ Run All ⌘ Restart ⌘ Clear All Outputs Variables Outline ... Python 3.10.11

```
# Function to predict the output
def predict(x1):
    input1 = x1
    input2 = x2

    # Forward pass
    ...
    node_1_output = input1 * w1 + input2 * w3 + bias1
    node_1_output = activation_ReLu(node_1_output)

    ...
    node_2_output = input1 * w2 + input2 * w4 + bias2
    node_2_output = activation_ReLu(node_2_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
```

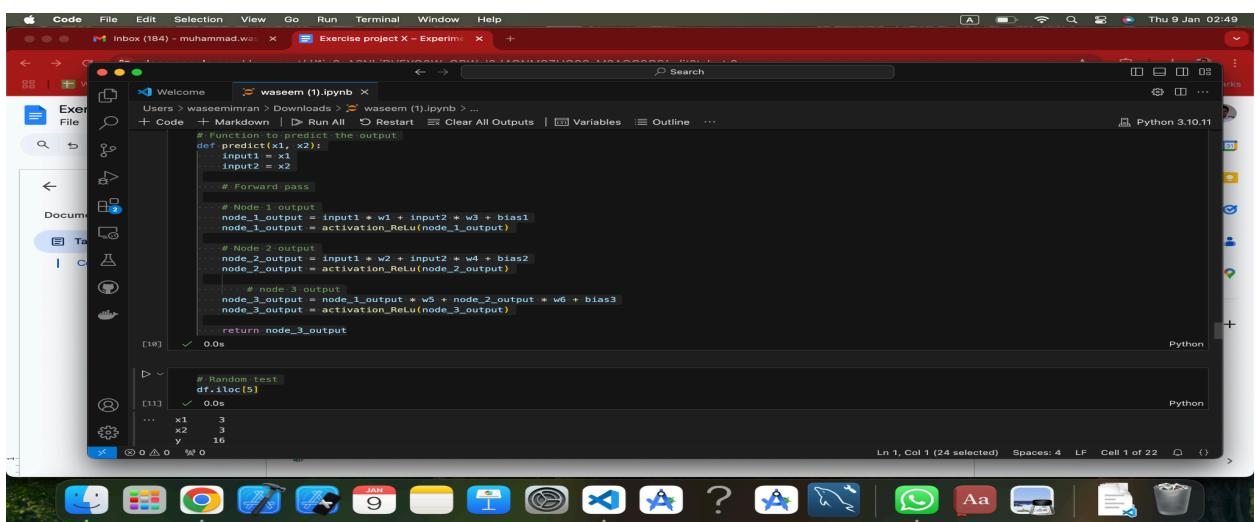
0.0s

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

0.0s

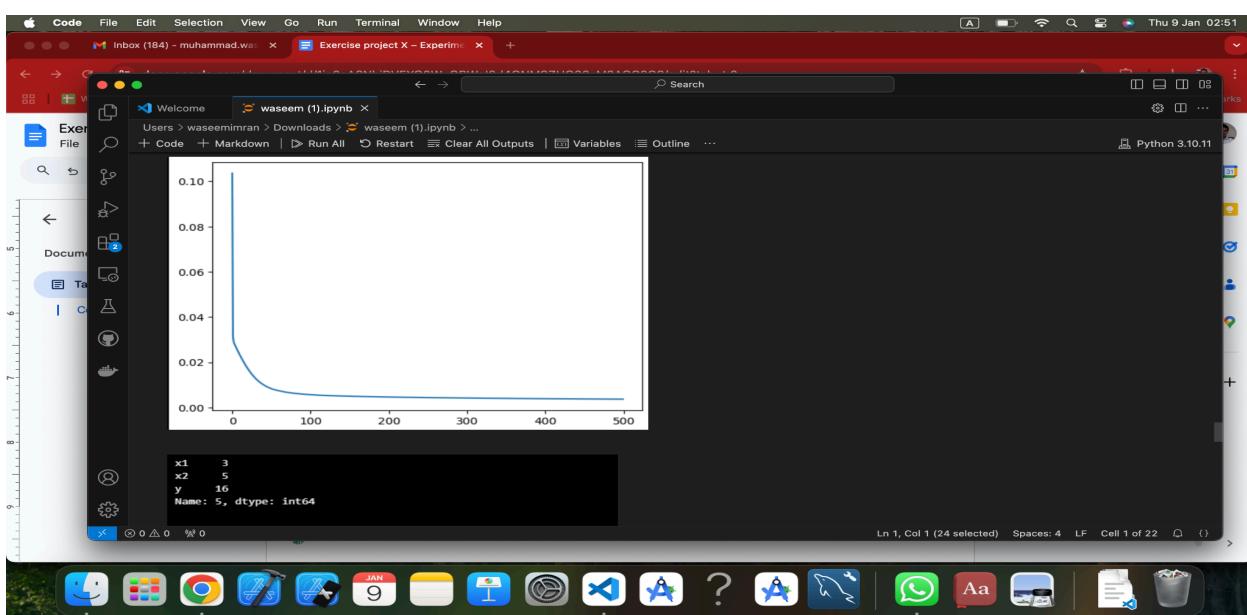
```
x1 3
x2 3
y 16
```

Ln 1, Col 1 (24 selected) Spaces: 4 LF Cell 1 of 22



A screenshot of a Mac OS X desktop showing a Jupyter Notebook interface. The notebook is titled 'waseem (1).ipynb'. The code cell contains the following Python code:

```
# 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  
array([0.05])  
  
# Finally scale the result back to the original scale  
df['y'].max() * result  
array([0.05])  
  
# Experiment with different values  
df['y'].max() * result  
array([0.781829167529744])
```



A screenshot of a Mac OS X desktop showing a Jupyter Notebook interface. The notebook is titled 'waseem (1).ipynb'. The code cell contains the following Python code:

```
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  
array([0.05])  
  
# Finally scale the result back to the original scale  
df['y'].max() * result  
array([0.05])  
  
16.52363390273419
```

