CSC 173 - Intelligent Systems

- Binary Classifier for Bathroom Quality Prediction
- Author: Michael James Carnaje
- Date: October 05, 2024

- Overview: This notebook implements a neural network to predict bathroom quality (Good/Bad) based on area and bathrooms.
- Data: house_bathroom.csv

We'll start by loading the libraries and the dataset.

- Pandas for data manipulation.
- Numpy for numerical operations.
- Matplotlib for visualization.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Load the dataset, I converted the table from the assignment file to a csv file.

In [ ]:
```
df = pd.read_csv('house_bathroom.csv')

df
```

Out[ ]:

	Area (sq ft)	Bathrooms	Classification
0	2104	3	Good
1	1600	3	Good
2	2400	3	Good
3	1416	2	Bad
4	3000	4	Bad
5	1985	4	Good
6	1534	3	Bad
7	1427	3	Good
8	1380	3	Good
9	1494	3	Good

Now we will convert the 'Good' and 'Bad' values in the Classification column to 1 and 0 respectively.

In [ ]:
```
df['Classification'] = df['Classification'].apply(lambda x: 1 if x == 'Good'

df
```

Out[ ]:

	Area (sq ft)	Bathrooms	Classification
0	2104	3	1
1	1600	3	1
2	2400	3	1
3	1416	2	0
4	3000	4	0
5	1985	4	1
6	1534	3	0
7	1427	3	1
8	1380	3	1
9	1494	3	1

So to answer the **second question**, we have:

- Input variables: Area (sq ft), Bathrooms
- Output variable: Classification

In [ ]:
```
inputX = df[['Area (sq ft)', 'Bathrooms']].values
inputY = df['Classification'].values
```

In [ ]:
```
inputX = np.array(inputX)
inputX
```

Out[ ]:
```
array([[2104,    3],
       [1600,    3],
       [2400,    3],
       [1416,    2],
       [3000,    4],
       [1985,    4],
       [1534,    3],
       [1427,    3],
       [1380,    3],
       [1494,    3]])
```

In [ ]:
```
inputY = np.array(inputY)
inputY
```

Out[ ]: array([1, 1, 1, 0, 0, 1, 0, 1, 1, 1])

Let's define the parameters for the binary classifier network.

- Learning rate: The learning rate is the step size that determines how much the weights and biases are updated during training.
- Training epochs: The number of epochs is the number of times the entire dataset is passed through the neural network during training.
- Display step: The display step is the number of epochs after which the loss is displayed.
- Number of samples: The number of samples is the total number of data points in the dataset.

In [ ]:
```
learning_rate = 0.01
training_epochs = 10000
display_step = 100
n_samples = inputX.size
```

So to answer the **5th question**, the learning rate is 0.01, the training epochs is 10000.

I used the sigmoid function as the **activation function** because it is a binary classifier. and this will answer the **4th question**.

In [ ]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

I used the mean squared error as the **loss function** because it is a regression problem. and this will answer the **4th question**.

In [ ]:
```
def mse_loss(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean()
```

In [ ]:
```
def derivative_sigmoid(x):
    fx = sigmoid(x)
    return fx * (1 - fx)
```

This BinaryClassifier is the same as the *'Machine Learning for Beginners: An Introduction to Neural Networks'* by **Victor Zhou**. But I made some modifications to the code by adding the *normalize_input* method to normalize the input data, which was not done in the example, and also added the saving of errors in a list for plotting later.

To answer the **3th question**, The initialization of the weights and biases are random, which I got from the example by **Victor Zhou**.

In [ ]:
```
class BinaryClassifier:
    def __init__(self, learning_rate, training_epochs, display_step):
```

```
        self.learning_rate = learning_rate
        self.training_epochs = training_epochs
        self.display_step = display_step

        # Normalize the input data
        self.input_mean = None
        self.input_std = None

        # Initialize weights
        self.w1 = np.random.normal()
        self.w2 = np.random.normal()
        self.w3 = np.random.normal()
        self.w4 = np.random.normal()
        self.w5 = np.random.normal()
        self.w6 = np.random.normal()

        # Initialize biases
        self.b1 = np.random.normal()
        self.b2 = np.random.normal()
        self.b3 = np.random.normal()

    def normalize_input(self, X):
        if self.input_mean is None or self.input_std is None:
            self.input_mean = np.mean(X, axis=0)
            self.input_std = np.std(X, axis=0)

        return (X - self.input_mean) / (self.input_std + 1e-8)


    def feedforward(self, x):
        x_normalized = (x - self.input_mean) / (self.input_std + 1e-8)

        h1 = sigmoid(self.w1 * x_normalized[0] + self.w2 * x_normalized[1] +
        h2 = sigmoid(self.w3 * x_normalized[0] + self.w4 * x_normalized[1] +
        o1 = sigmoid(self.w5 * h1 + self.w6 * h2 + self.b3)
        return o1


    def train(self, data, all_y_trues):
        normalized_data = self.normalize_input(data)


        errors = []

        for epoch in range(self.training_epochs):
            epoch_error = 0
            for x, y_true in zip(normalized_data, all_y_trues):

                sum_h1 = self.w1 * x[0] + self.w2 * x[1] + self.b1
                h1 = sigmoid(sum_h1)

                sum_h2 = self.w3 * x[0] + self.w4 * x[1] + self.b2
                h2 = sigmoid(sum_h2)
```

```
            sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
            o1 = sigmoid(sum_o1)
            y_pred = o1

            d_L_d_ypred = -2 * (y_true - y_pred)

            d_ypred_d_w5 = h1 * derivative_sigmoid(sum_o1)
            d_ypred_d_w6 = h2 * derivative_sigmoid(sum_o1)
            d_ypred_d_b3 = derivative_sigmoid(sum_o1)

            d_ypred_d_h1 = self.w5 * derivative_sigmoid(sum_o1)
            d_ypred_d_h2 = self.w6 * derivative_sigmoid(sum_o1)

            d_h1_d_w1 = x[0] * derivative_sigmoid(sum_h1)
            d_h1_d_w2 = x[1] * derivative_sigmoid(sum_h1)
            d_h1_d_b1 = derivative_sigmoid(sum_h1)

            d_h2_d_w3 = x[0] * derivative_sigmoid(sum_h2)
            d_h2_d_w4 = x[1] * derivative_sigmoid(sum_h2)
            d_h2_d_b2 = derivative_sigmoid(sum_h2)

            self.w1 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h1 *
            self.w2 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h1 *
            self.w3 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h2 *
            self.w4 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h2 *
            self.w5 -= self.learning_rate * d_L_d_ypred * d_ypred_d_w5
            self.w6 -= self.learning_rate * d_L_d_ypred * d_ypred_d_w6
            self.b1 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h1 *
            self.b2 -= self.learning_rate * d_L_d_ypred * d_ypred_d_h2 *
            self.b3 -= self.learning_rate * d_L_d_ypred * d_ypred_d_b3

            loss = mse_loss(y_true, y_pred)
            epoch_error += loss

        avg_epoch_error = epoch_error / len(data)
        errors.append(avg_epoch_error)

        if epoch % self.display_step == 0:
            print(f"Epoch: {epoch}/{self.training_epochs}, Loss: {avg_ep

    return errors
```

In [ ]:

```
network = BinaryClassifier(learning_rate, training_epochs, display_step)
errors = network.train(inputX, inputY)

final_predictions = np.array([network.feedforward(x) for x in inputX])
final_cost = mse_loss(inputY, final_predictions)

print(f"Final cost of the network: {final_cost:.6f}")
```

```
Epoch: 0/10000, Loss: 0.30775734242759567
Epoch: 100/10000, Loss: 0.22010491626192982
Epoch: 200/10000, Loss: 0.2157565425127642
```

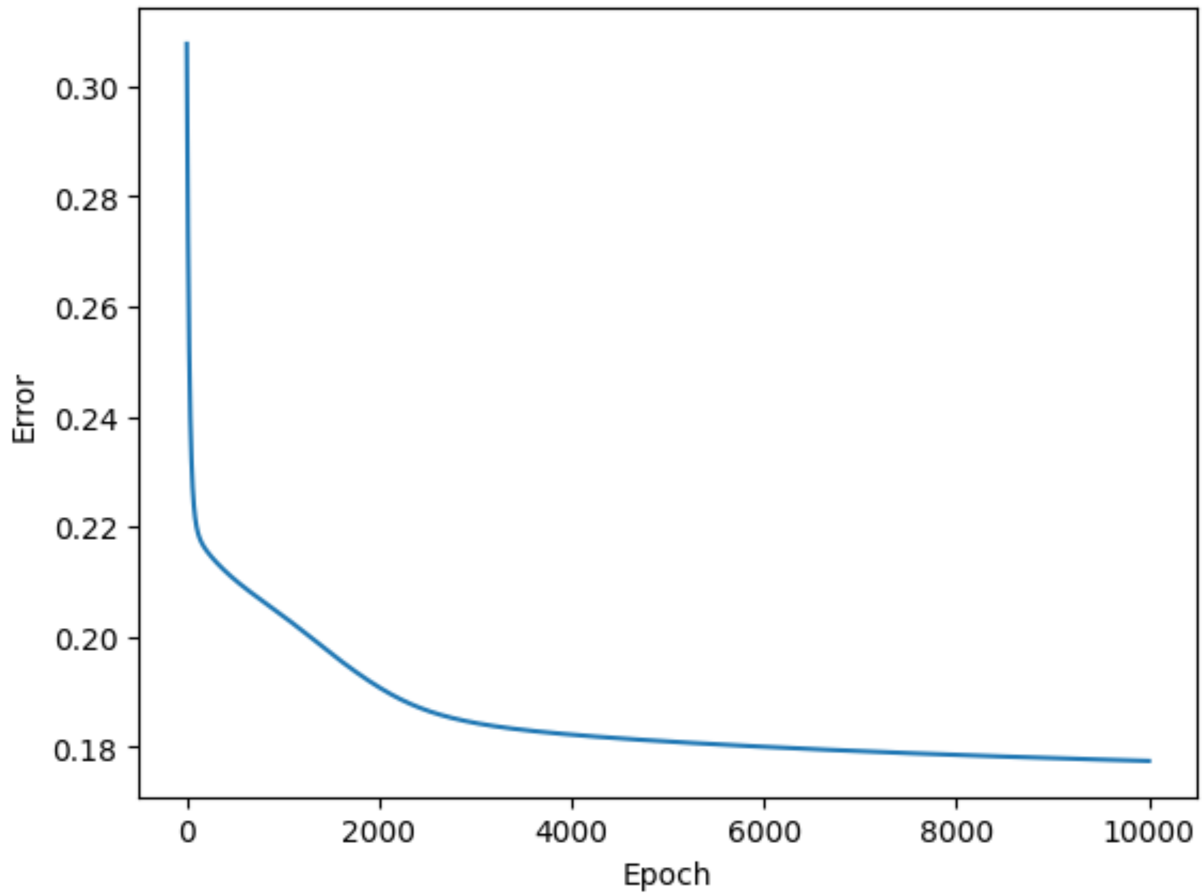
```
Epoch: 300/10000, Loss: 0.21364759181360266
Epoch: 400/10000, Loss: 0.21188570350048538
Epoch: 500/10000, Loss: 0.21032802827455552
Epoch: 600/10000, Loss: 0.20891134371324585
Epoch: 700/10000, Loss: 0.20758374243632266
Epoch: 800/10000, Loss: 0.2063023712387197
Epoch: 900/10000, Loss: 0.20503301523972234
Epoch: 1000/10000, Loss: 0.20374974688304529
Epoch: 1100/10000, Loss: 0.20243561671994445
Epoch: 1200/10000, Loss: 0.20108467312109904
Epoch: 1300/10000, Loss: 0.19970402106430502
Epoch: 1400/10000, Loss: 0.19831281519638477
Epoch: 1500/10000, Loss: 0.19693576932877568
Epoch: 1600/10000, Loss: 0.19559382756508173
Epoch: 1700/10000, Loss: 0.19429905101212008
Epoch: 1800/10000, Loss: 0.19305740715459146
Epoch: 1900/10000, Loss: 0.19187530811076808
Epoch: 2000/10000, Loss: 0.19076307890790126
Epoch: 2100/10000, Loss: 0.18973299955853984
Epoch: 2200/10000, Loss: 0.1887948905905144
Epoch: 2300/10000, Loss: 0.18795313121704568
Epoch: 2400/10000, Loss: 0.1872063772922208
Epoch: 2500/10000, Loss: 0.18654896138848973
Epoch: 2600/10000, Loss: 0.18597264339336478
Epoch: 2700/10000, Loss: 0.18546804495022676
Epoch: 2800/10000, Loss: 0.1850256427223406
Epoch: 2900/10000, Loss: 0.18463639235190182
Epoch: 3000/10000, Loss: 0.18429207919319165
Epoch: 3100/10000, Loss: 0.18398547621320857
Epoch: 3200/10000, Loss: 0.183710372932582 5
Epoch: 3300/10000, Loss: 0.18346152500437116
Epoch: 3400/10000, Loss: 0.1832345609480828
Epoch: 3500/10000, Loss: 0.1830258710675099
Epoch: 3600/10000, Loss: 0.18283249436152735
Epoch: 3700/10000, Loss: 0.1826520124655962
Epoch: 3800/10000, Loss: 0.1824824550602523
Epoch: 3900/10000, Loss: 0.1823222182748293
Epoch: 4000/10000, Loss: 0.1821699959162732
Epoch: 4100/10000, Loss: 0.18202472246138687
Epoch: 4200/10000, Loss: 0.1818855263658898
Epoch: 4300/10000, Loss: 0.18175169215716447
Epoch: 4400/10000, Loss: 0.18162262985319824
Epoch: 4500/10000, Loss: 0.181497850403126
Epoch: 4600/10000, Loss: 0.18137694602474325
Epoch: 4700/10000, Loss: 0.1812595744935706
Epoch: 4800/10000, Loss: 0.1811454466025137
Epoch: 4900/10000, Loss: 0.18103431615506474
Epoch: 5000/10000, Loss: 0.18092597197711363
Epoch: 5100/10000, Loss: 0.18082023153395668
Epoch: 5200/10000, Loss: 0.1807169358222624
Epoch: 5300/10000, Loss: 0.1806159452741852
Epoch: 5400/10000, Loss: 0.18051713646507064
Epoch: 5500/10000, Loss: 0.18042039945959792
```

```
Epoch: 5600/10000, Loss: 0.18032563566576493
Epoch: 5700/10000, Loss: 0.18023275609356199
Epoch: 5800/10000, Loss: 0.18014167993688596
Epoch: 5900/10000, Loss: 0.1800523334143936
Epoch: 6000/10000, Loss: 0.17996464881849128
Epoch: 6100/10000, Loss: 0.17987856373227074
Epoch: 6200/10000, Loss: 0.1797940203825214
Epoch: 6300/10000, Loss: 0.1797109651034587
Epoch: 6400/10000, Loss: 0.17962934789088378
Epoch: 6500/10000, Loss: 0.17954912203044843
Epoch: 6600/10000, Loss: 0.17947024378676146
Epoch: 6700/10000, Loss: 0.17939267214245996
Epoch: 6800/10000, Loss: 0.17931636857820732
Epoch: 6900/10000, Loss: 0.17924129688601528
Epoch: 7000/10000, Loss: 0.17916742300940386
Epoch: 7100/10000, Loss: 0.17909471490479614
Epoch: 7200/10000, Loss: 0.17902314241925335
Epoch: 7300/10000, Loss: 0.17895267718024416
Epoch: 7400/10000, Loss: 0.17888329249364116
Epoch: 7500/10000, Loss: 0.17881496324658552
Epoch: 7600/10000, Loss: 0.17874766581227383
Epoch: 7700/10000, Loss: 0.1786813779541161
Epoch: 7800/10000, Loss: 0.178616078727104
Epoch: 7900/10000, Loss: 0.17855174837461166
Epoch: 8000/10000, Loss: 0.1784883682192349
Epoch: 8100/10000, Loss: 0.17842592054665318
Epoch: 8200/10000, Loss: 0.178364388481865
Epoch: 8300/10000, Loss: 0.1783037558574938
Epoch: 8400/10000, Loss: 0.17824400707418223
Epoch: 8500/10000, Loss: 0.17818512695337158
Epoch: 8600/10000, Loss: 0.17812710058299408
Epoch: 8700/10000, Loss: 0.17806991315677984
Epoch: 8800/10000, Loss: 0.17801354980798323
Epoch: 8900/10000, Loss: 0.17795799543836816
Epoch: 9000/10000, Loss: 0.17790323454324725
Epoch: 9100/10000, Loss: 0.17784925103324237
Epoch: 9200/10000, Loss: 0.17779602805323283
Epoch: 9300/10000, Loss: 0.1777435477986765
Epoch: 9400/10000, Loss: 0.17769179132913282
Epoch: 9500/10000, Loss: 0.17764073837839797
Epoch: 9600/10000, Loss: 0.17759036716017632
Epoch: 9700/10000, Loss: 0.17754065416767134
Epoch: 9800/10000, Loss: 0.17749157396488974
Epoch: 9900/10000, Loss: 0.1774430989668072
Final cost of the network: 0.177158
```

To answer the **7th question**, I save the errors in a list and plot it.

In [ ]:

```
plt.plot(errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()
```



To answer the **8th question**, we will test the network with the sample properties which are not in the training data.

In [ ]:
```
sample_properties = np.array([[4000, 1], [1500, 3], [2000, 4]])

for sample in sample_properties:
    prediction = network.feedforward(sample)
    print(f"Area: {sample[0]}, Bathrooms: {sample[1]}, Prediction: {'Good' i
```

```
Area: 4000, Bathrooms: 1, Prediction: Bad (Probability: 0.4434)
Area: 1500, Bathrooms: 3, Prediction: Good (Probability: 0.7960)
Area: 2000, Bathrooms: 4, Prediction: Good (Probability: 0.9048)
```

References:

- Alammar, J. (2019, August 27). A visual and interactive guide to the basics of neural networks. The Blog of Jay Alammar. https://jalammar.github.io/visual-interactive-guide-basics-neural-networks/
- Zhou, V. (2018, September 12). Intro to neural networks: A beginner-friendly introduction. Victor Zhou. https://victorzhou.com/blog/intro-to-neural-networks/