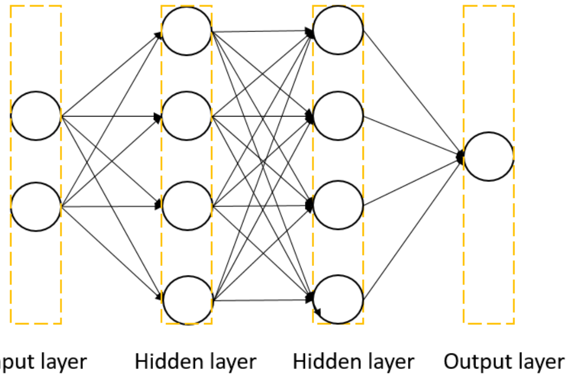
Summer 2022 Deep Learning

Report of Lab #1

Student name: 劉子齊 Jonathan
Student ID: 311605004

Part 1: Introduction

In this lab, my task is to implement a simple neural network without common frameworks, such as "TensorFlow" or "PyTorch", but through "NumPy" and other standard libraries. Also, the simple neural network will be implemented with forward pass and backpropagation using two hidden layers, as the structure shown in the following figure.



Besides, I would like to introduce the dataset I generated for this lab. As the instruction recommended, I generated the dataset through the given functions as below.

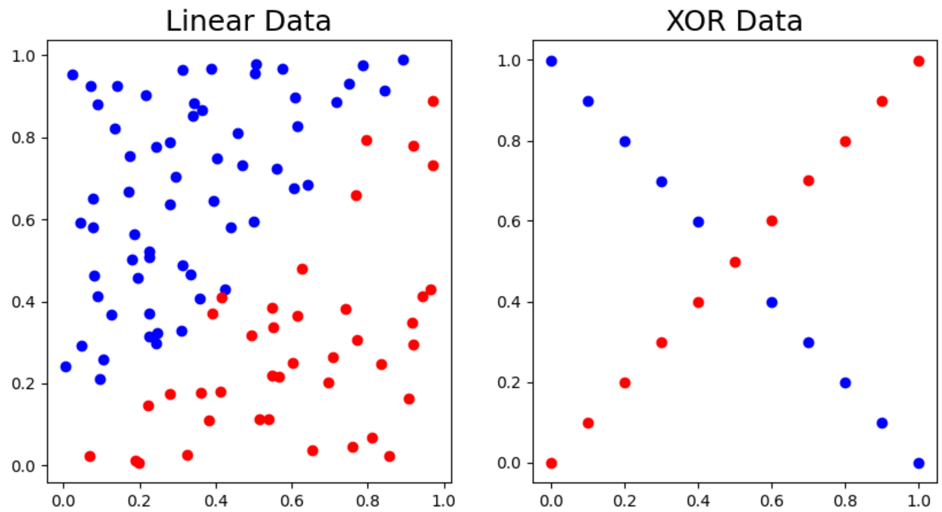
```python
# Data generation
def generate_linear(n=100):
    pts = np.random.uniform(0, 1, (n, 2))
    inputs = []
    labels = []

    for pt in pts:
        inputs.append([pt[0], pt[1]])
        distance = (pt[0] - pt[1]) / 1.414

        if pt[0] > pt[1]:
            labels.append(0)
        else:
            labels.append(1)

    return np.array(inputs), np.array(labels).reshape(n, 1)
```

```python
def generate_XOR_easy():
    inputs = []
    labels = []

    for i in range(11):
        inputs.append([0.1*i, 0.1*i])
        labels.append(0)

        if 0.1*i == 0.5:
            continue

        inputs.append([0.1*i, 1 - 0.1*i])
        labels.append(1)

    return np.array(inputs), np.array(labels).reshape(21, 1)
```

To make sure the data generated reached the required distributions, I constructed the following "show_data" function to show the distribution of the data I generated.

```python
# ========================================================
# Showing data

def show_data(X, Y, T):
    n = len(X)
    plt.figure(figsize = (5*n, 5))

    for i, x, y, t in zip(range(n), X, Y, T):
        y = np.round(y)
        plt.subplot(1, n, i+1)
        plt.title(t, fontsize = 18)
        for i in range(x.shape[0]):
            if y[i] == 0:
                plt.plot(x[i][0], x[i][1], 'ro')
            else:
                plt.plot(x[i][0], x[i][1], 'bo')

    plt.show()
# ========================================================
# Generate data now

x1, y1 = generate_linear(n=100)
x2, y2 = generate_XOR_easy()
show_data([x1,x2], [y1,y2], ['Linear Data', 'XOR Data'])

# ========================================================
```

For each experiment conducted, I will show the distribution of the data generated for the experiment in every experiment. The figure below was the data distribution of one of the experiments conducted, which is similar and obviously reached the requirements of data generation.



Part 2: Experiment setups

Part 2-A: Sigmoid Functions

In this lab, as the instruction recommended, I took Sigmoid function as the activation function, which is derived as the following mathematical notation.

Sigmoid function $\quad \sigma(x) = \frac{1}{1+e^{-x}}$

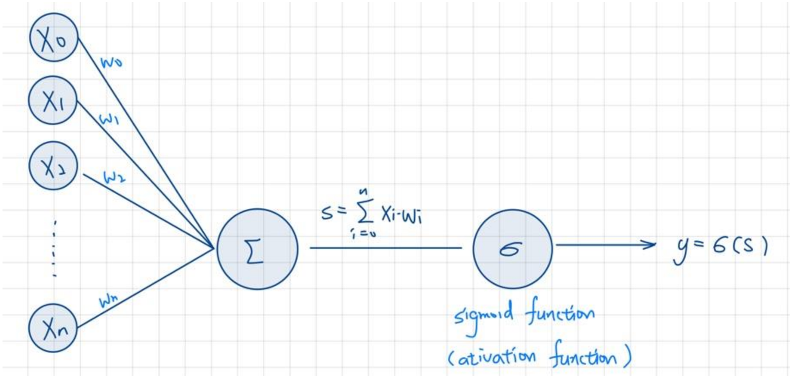
Derivation of $\sigma(x)$ $\quad \sigma'(x) = \frac{1}{1+e^{-x}} \cdot \frac{d}{dx}$

$$= \frac{d(1+e^{-x})^{-1}}{dx} = -(1+e^{-x})^{-2} \cdot \frac{d}{dx}(1+e^{-x})$$

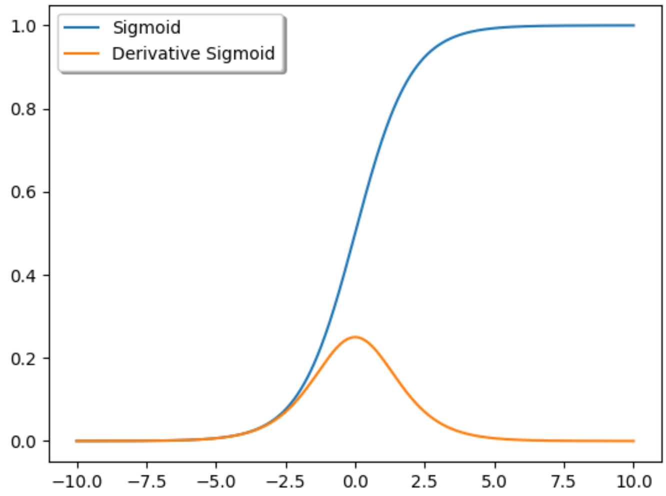
$$= -(1+e^{-x})(1+e^{-x})(-e^{-x})$$

$$= \sigma(x)(1-\sigma(x))$$

The figure below shows the role of the activation function, which is the sigmoid function in this case, plays in the layer of the neural network.



The following figure is my implementation of the sigmoid function taking the TA's hint as a reference. Besides, to take a deeper peek into sigmoid functions, I also drew it out through matplotlib, and the result is shown in the figure below.




```python
# ========================================================
# Sigmoid functions

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)

# check the sigmoid function
xs = np.linspace(-10, 10, 500)
plt.plot(xs, sigmoid(xs), label = 'Sigmoid')
plt.plot(xs, derivative_sigmoid(sigmoid(xs)), label = 'Derivative Sigmoid')
plt.legend(loc='upper left', shadow=True)
plt.show()

# ========================================================
```

Part 2-B: Neural Network

For the main part of this lab, which is the neural network, I divided it into two parts, the unit calculation part, and the neural network part.

In the unit calculation part, this is where the input vector "x" gets the output scalar "y". To multiply the output scalar "y", I extend the unit calculator which will be the neural network. The way I implemented the unit calculator is as the following capture of the "unit_calculator" function.

```python
# ========================================================
# Unit Calculator

class unit_calculation():
    def __init__(self, inputsize, outputsize):
        self.w = np.random.normal(0, 1, (inputsize + 1, outputsize))

    def forward(self, x):
        x = np.append(x, np.ones((x.shape[0], 1)), axis = 1)
        self.forward_grad = x
        self.y = sigmoid(np.matmul(x, self.w))
        return self.y

    def backward(self, der_c):
        self.backward_grad = np.multiply(derivative_sigmoid(self.y), der_c)
        return np.matmul(self.backward_grad, self.w[ : -1].T)

    def update(self, learning_rate):
        self.gradient = np.matmul(self.forward_grad.T, self.backward_grad)
        self.w = self.w - learning_rate * self.gradient
        return self.gradient

# ========================================================
```

With the unit calculator function above, next we'll move onto the main part, which is the neural network extended through the unit calculator. As shown in the following figure, this is how I implemented the neural network. Like I mentioned earlier, a unit calculator can only output a scalar, I put N units in the layers to output N scalars.

```python
# ========================================================
# Neural Network

class NN():
    # initialize the weight matrix
    def __init__(self, size, learning_rate):
        self.learning_rate = learning_rate
        self.layers = []

        for a, b in zip(size, (size[1:] + [0])):
            if (a+1)*b != 0:
                self.layers += [unit_calculation(a, b)]

    def forward(self, x):
        new_x = x
        for layer in self.layers:
            new_x = layer.forward(new_x)
        return new_x

    def backward(self, der_cost):
        new_der_cost = der_cost
        for layer in self.layers[::-1]:
            new_der_cost = layer.backward(new_der_cost)

    def update(self):
        gradients = []
        for layer in self.layers:
            gradients = gradients + [layer.update(self.learning_rate)]
        return gradients

# ========================================================
```

Part 2-C: Backpropagation

As we all know, backpropagation is to update the weight matrix. In the beginning, I initialized all the weight parameters in the network randomly. The goal here in backpropagation is the minimize the cost from the loss function below.

For the loss function, I chose to use the Mean Square Error, which was mentioned in class, as my loss function. In the mathematical perspective, I implemented it as below.

$$Loss(y.\hat{y}) = MSE(y,\hat{y}) = E((y-\hat{y})^2) = \frac{\Sigma(y-\hat{y})^2}{N}$$

$$Loss'(y.\hat{y}) = \frac{\partial E((y-\hat{y})^2)}{\partial y} = \frac{1}{N}\cdot\left(\frac{\partial(y-\hat{y})^2}{\partial y}\right)$$

$$= \frac{1}{N}\left(2(y-\hat{y})\cdot\frac{\partial(y-\hat{y})}{\partial y}\right) = \frac{2}{N}(y-\hat{y})$$

Through the derived function, I implemented by the way in the figure below.

```
# ========================================================
# MSE - Loss Function

def mse_loss(y, yhat):
    return np.mean((y - yhat)**2)

def derivative_mse_loss(y, yhat):
    return (y - yhat)*(2 / y.shape[0])

# ========================================================
```

With the "mse_loss" and the "derivative_mse_loss" functions defined, our goal is to minimize the cost. We already know that we update the network weight through gradient descent. We can see the mathematical computation in the figure below.

C is cost here

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\cdot\frac{\partial C}{\partial z} \quad \text{(Chain Rule)}$$

By forward gradient $\Rightarrow \frac{\partial z}{\partial w} = \frac{\partial x'\cdot w}{\partial w} = x'$

By backward gradient $\Rightarrow \frac{\partial C}{\partial z} = \frac{\partial y}{\partial z}\cdot\frac{\partial C}{\partial y}$

$$= \sigma'(z)\cdot\frac{\partial C}{\partial y} \quad (y=\sigma(z))$$

we know C comes from $L(y,\hat{y})$

$\Rightarrow \frac{\partial C}{\partial y} = L'(y,\hat{y})$

in this case

$$\frac{\partial C}{\partial y_n} = \frac{\partial z_{n+1}}{\partial y_n}\cdot\frac{\partial C}{\partial z_{n+1}}$$

$$\frac{\partial z_{n+1}}{\partial y_n} = W^T_{n+1} \Rightarrow z_{n+1} = y_n\cdot W_{n+1}$$

Through the mathematical calculation above, we can see the output y of the current layer is the input for the next layer. In the final “purple” formula, we get that we compute the output layer first, then send the parameters to the previous layer. Hence, we can compute $\frac{\partial C}{\partial z}$ in every layer.

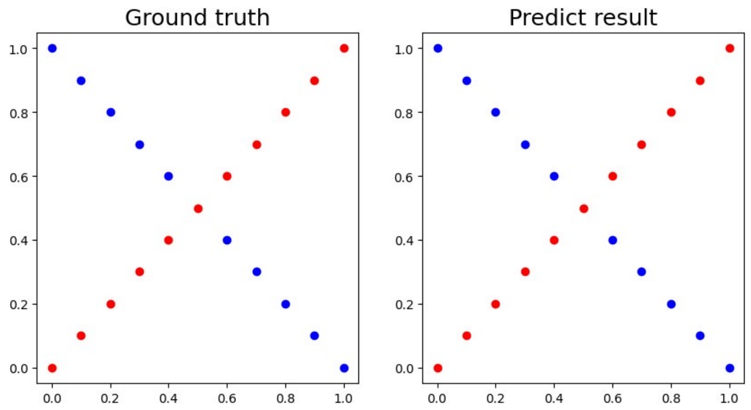
Part 3: Results of my testing

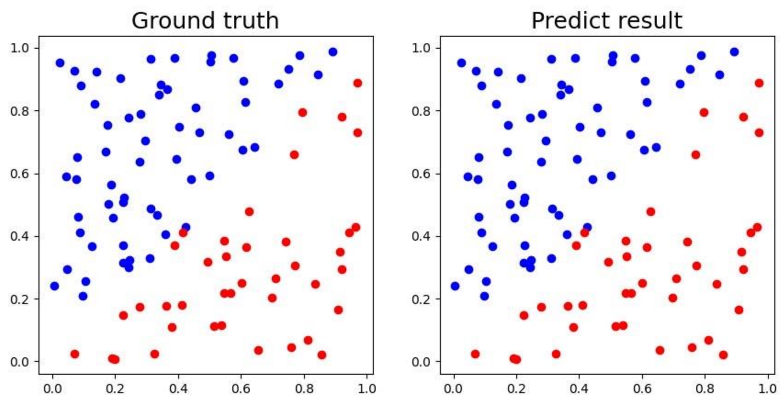
Part 3-0: The way I run the NN for testing

The following figure is how I implemented the NNs and run the testing, I will show how I drew the comparison figures and show the results in the following sections.

```
# ========================================================
# testing

linear_nn = NN([2, 4, 4, 1], 0.1)
XOR_nn = NN([2, 4, 4, 1], 0.1)

epoches = 100000
threshold = 0.005

linear_stop = False
XOR_stop = False

linear_x, linear_y = generate_linear()
XOR_x, XOR_y = generate_XOR_easy()
show_data([linear_x, XOR_x], [linear_y, XOR_y], ['Linear Data', 'XOR Data'])

# store data to draw the learning curve
linear_loss_data = []
xor_loss_data = []
epoch_data = []

for epoch in range(epoches):
    if not linear_stop:
        y1 = linear_nn.forward(linear_x)
        linear_loss = mse_loss(y1, linear_y)
        linear_nn.backward(derivative_mse_loss(y1, linear_y))
        linear_nn.update()

        if linear_loss < threshold:
            linear_stop = True
            print("Linear goal accomplished...:)")

    if not XOR_stop:
        y2 = XOR_nn.forward(XOR_x)
        XOR_loss = mse_loss(y2, XOR_y)
        XOR_nn.backward(derivative_mse_loss(y2, XOR_y))
        XOR_nn.update()

        if XOR_loss < threshold:
            XOR_stop = True
            print("XOR goal accomplished...:)")
```

Part 3-A: Screenshot and comparison figure





The following function in the figure is how I get the comparison plots.

```
# ========================================================
# Showing result

def show_result(x, y, pred_y):
    plt.subplot(1, 2, 1)
    plt.title('Ground truth', fontsize = 18)
    for i in range(x.shape[0]):
        if y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.subplot(1, 2, 2)
    plt.title('Predict result', fontsize = 18)
    pred_y = np.round(pred_y)
    for i in range(x.shape[0]):
        # print('pred_y: ', pred_y[i])
        if pred_y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.show()

# ========================================================
```

```
# show the result comparison figure
show_result(linear_x, linear_y, y1)
show_result(XOR_x, XOR_y, y2)
```

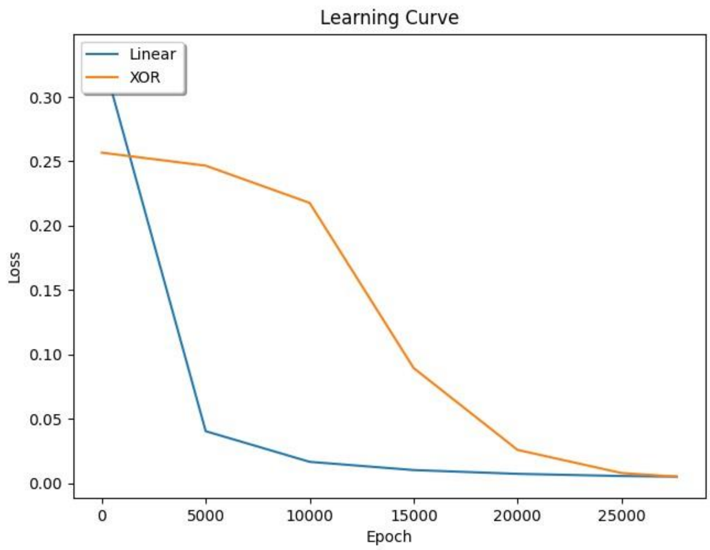
Part 3-B: Show the accuracy of your prediction

```
Linear test loss :  0.005
Linear test result :
 [[1.   ]
 [0.   ]
 [1.   ]
 [0.001]
 [0.983]
 [0.999]
 [0.   ]
 [0.   ]
 [0.999]
 [1.   ]
 [0.   ]
 [0.971]
 [0.133]
 [0.954]
 [1.   ]
 [0.   ]
 [1.   ]
 [1.   ]
 [0.999]
 [0.991]
 [1.   ]
 [0.   ]
 [0.999]
 [0.807]
 [0.999]
 [0.654]
 [0.001]
 [0.999]
 [0.   ]
 [0.99 ]
 [0.   ]
 [0.001]
 [0.   ]
 [0.   ]
 [0.751]
 [0.999]
 [0.   ]
 [0.001]
 [1.   ]
 [0.998]
 [0.003]
 [0.905]
 [0.999]
 [1.   ]
 [0.003]
 [0.865]
 [0.998]
 [0.   ]
 [0.143]
 [0.   ]
 [1.   ]
 [0.999]
 [0.001]
 [0.001]
 [0.   ]
 [0.012]
 [0.055]
 [0.   ]
 [0.994]
 [0.999]
 [0.222]
 [1.   ]
 [0.   ]
 [0.002]
 [0.001]
 [0.008]
 [0.003]
 [0.999]
 [1.   ]
 [0.934]
 [0.263]
 [0.001]
 [1.   ]
 [0.   ]
 [1.   ]
 [0.   ]
 [1.   ]
 [0.   ]
 [1.   ]
 [1.   ]
 [0.927]
 [1.   ]
 [1.   ]
 [0.   ]
 [0.997]
 [0.985]
 [0.   ]
 [0.999]
 [0.241]
 [0.077]
 [0.   ]
 [0.   ]
 [0.01 ]
 [1.   ]
 [0.004]
 [1.   ]
 [0.   ]
 [0.126]
 [0.   ]
 [0.985]]

XOR test loss :  0.005
XOR test result :
 [[0.0144]
 [0.9998]
 [0.0068]
 [0.9998]
 [0.0076]
 [0.9996]
 [0.0243]
 [0.9971]
 [0.1002]
 [0.8232]
 [0.1628]
 [0.1054]
 [0.8486]
 [0.0432]
 [0.9966]
 [0.017 ]
 [0.9982]
 [0.0078]
 [0.9978]
 [0.0042]
 [0.9968]]
```

The following figure is the way I obtain and print the result.

```python
print('\nLinear test loss : ', mse_loss(y1, linear_y).round(5))
print('Linear test result : \n', y1.round(3))

print('XOR test loss : ', mse_loss(y2, XOR_y).round(5))
print('XOR test result : \n', y2.round(4))
```

Part 3-C: Learning curve (loss, epoch curve)



The following figure is the way I obtain and plot the learning curve.

```python
# store data to draw the learning curve
linear_loss_data = []
xor_loss_data = []
epoch_data = []
```

```python
    if epoch % 5000 == 0:
        print('epoch {:4d} linear loss : {:.4f} \t XOR loss : {:.4f}'.format(epoch, linear_loss, XOR_loss))

        linear_loss_data.append(linear_loss)
        xor_loss_data.append(XOR_loss)
        epoch_data.append(epoch)

    if linear_stop and XOR_stop:
        print('epoch {:4d} linear loss : {:.4f} \t XOR loss : {:.4f}'.format(epoch, linear_loss, XOR_loss))
        linear_loss_data.append(linear_loss)
        xor_loss_data.append(XOR_loss)
        epoch_data.append(epoch)
        break
```

```python
plt.plot(epoch_data, linear_loss_data, label = 'Linear')
plt.plot(epoch_data, xor_loss_data, label = 'XOR')
plt.legend(loc='upper left', shadow=True)
plt.title("Learning Curve")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

Part 4: Discussion

Part 4-A: Try different learning rates

In this section, I' ll try on different learning rates on the same dataset, and I' ll take record of the epochs needed for the neural networks to reach the target loss, which is 0.005, on two kinds of datasets. The following chart is the result of the experiments using different learning rates.

Learning Rate	Linear Data	XOR Data
0.001	3,172,986 epochs	2,896,658 epochs
0.01	679,327 epochs	359,669 epochs
0.1	36,717 epochs	30,658 epochs
0.5	10,442 epochs	5,910 epochs
0.999	1,376 epochs	3,394 epochs

Through the experiments above, we can easily find the epochs required for the neural networks to reach the loss goal increases as we lower the learning rate, which proves that higher learning rate requires less training time in the same background conditions.

Part 4-B: Try different numbers of hidden units

In this part, I' ll also test the results of different numbers of hidden units in the same background condition, and see how the different neural networks preform in reaching the requiring loss, which is 0.005. The following chart is the result of the experiments using different hidden units.

Note | The learning rates in this part are all set to 0.1

Hidden Units	Linear Data	XOR Data
2	144,633 epochs	1,000,000+ epochs
4	36,717 epochs	30,658 epochs
8	24,257 epochs	28,231 epochs
16	14,090 epochs	18,868 epochs
32	12,682 epochs	30,570 epochs

From the result above, we can find the neural network starts to perform a bit strange when there are 2 or 32 hidden units in the hidden layers. In the case of this lab, the complexity of the dataset is actually not too high. Hence, we can find the layer with 16 hidden units performs the best among the experiments conducted. When the number of hidden units reached 32, the neural network started to act not so good, or sometimes even worse, on the datasets. In my opinion, 16 hidden units in a layer will be the best number in this lab.

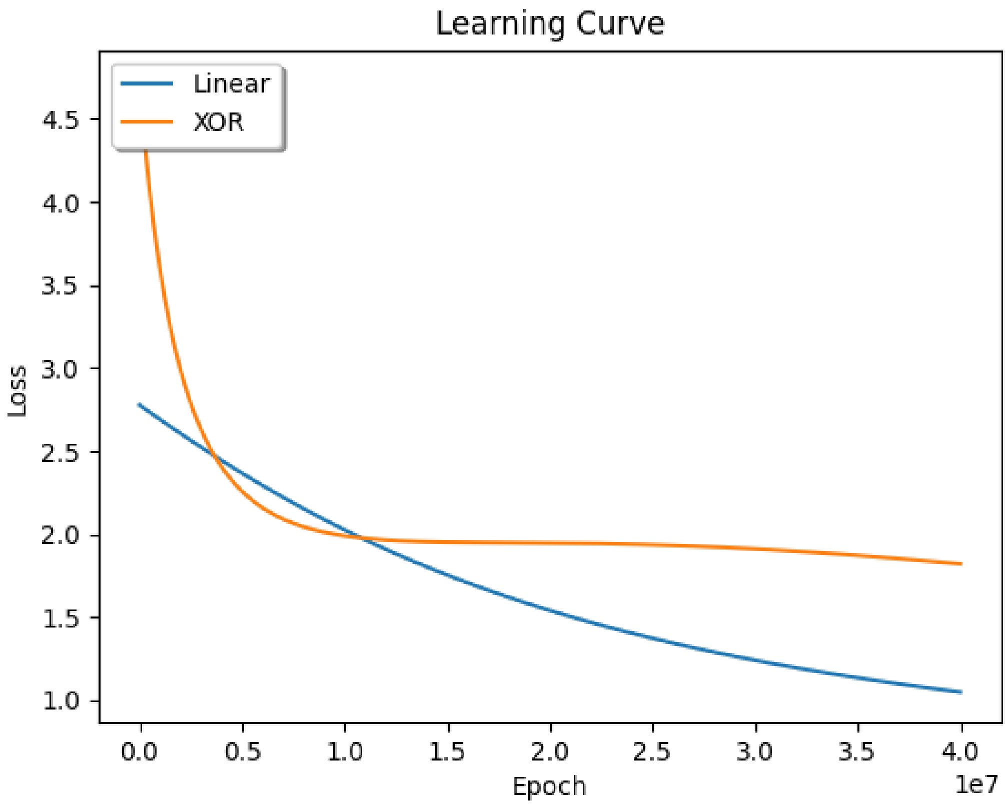
Part 4-C: Try without activation functions

If we took the activation function away, the weight and the bias will simple do linear transformation when updating. As mentioned in class, linear equation might be easier to solve, but it might also loose the capacity to solve problems with higher complexity. If the neural network got no activation functions in it, then it will become just a linear regression model. The main work activation functions do is to make the input capable to learn and perform tasks with higher complexity.

For this part, I changed the sigmoid functions by outputting the input directly, which is shown in the figure below.

```
# ========================================================
# Sigmoid functions

def sigmoid(x):
    # return 1.0 / (1.0 + np.exp(-x))
    return x

def derivative_sigmoid(x):
    # return np.multiply(x, 1.0 - x)
    return x
```

After taking the activation functions away, I found the original learning rate make the model's loss overflow. Hence, I started to lower the learning rate to avoid gradient exploration. After testing nine learning rates, without exploding during the process, I found myself able to get the proper result not until I lower to learning rate to "`0.000000001`". However, the model still can reach the expected result, which if the loss with 0.005, even with $4 * 10^7$ epochs. The following figure shows the learning curve of the model.



Through this experiment, we can find the importance of activation functions if we want to work on more complicated cases. Without activation functions, it is nearly impossible for us to be able to perform complex tasks, which might cost us plenty of time without getting the expecting result.