Lab 1 Back Propagation

109550121 温柏萱

Introduction

A fully connected neural network consists of layers of neurons that are able to perform matrix multiplications. Its operation can be understood as the weighted sum of the inputs, where the weights are adjusted through training in order for the neural network to generate the desired output corresponding to the input. The calculation can be separated into two phases: the forward pass and the

backward pass. The forward pass feeds the data into the neural network and obtains the result. The loss is obtained by comparing the result with the ground truth. To modify the weight matrices and biases, we perform the backward pass (backpropagation), where we take the values that minimize the loss to update the weights. According to the Extreme Value Theorem, the local extreme values occur where the gradients are zero. Therefore, the backward pass modifies the weight matrix according to the gradient of the loss function from the previous layer with the intention of achieving minimal loss.

To wrap up, the neural network iteratively modifies its weight matrices until the difference between the output and the ground truth falls below a predefined threshold.

Experiment Setups

Sigmoid Functions

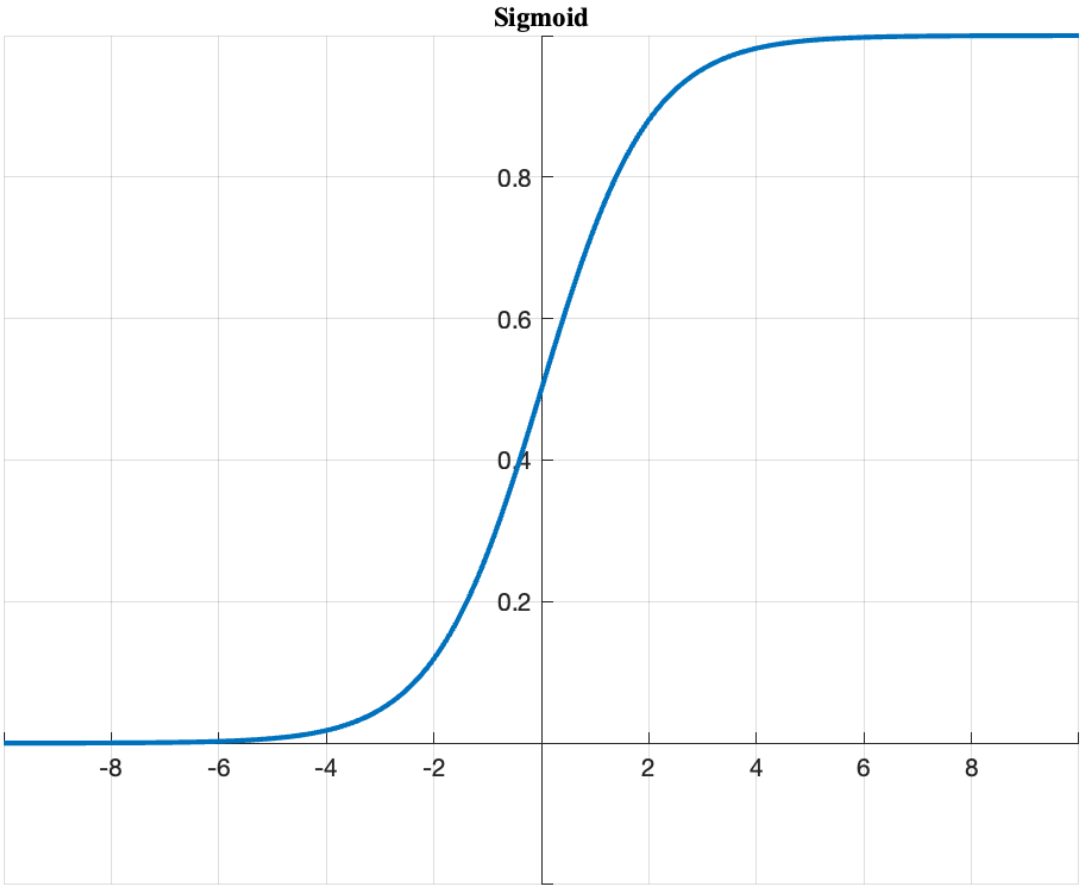
The sigmoid function is a common activation function that takes the input value and map its value into the range $[0, 1]$, this characteristic makes it a popular choice when dealing with linear classification problems, with the ability to convert input signals into a form that approximates probabilities. However, when the input values are unpleasently large or small, the output values of the Sigmoid function staturates to 1 or -1, limiting its sensitivy to the input.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1}$$

Its first-order derivative is derived as follows

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}(\frac{1}{1+e^{-x}}) = \frac{d}{dx}(1+e^{-x})^{-1} \\
&= \frac{d(1+e^{-x})^{-1}}{d(1+e^{-x})} \cdot \frac{d(1+e^{-x})}{de^{-x}} \cdot \frac{de^{-x}}{d(-x)} \cdot \frac{d(-x)}{dx} \\
&= -(1+e^{-x})^{-2} \cdot 1 \cdot e^{-x} \cdot (-1) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}(\frac{e^{-x}}{1+e^{-x}}) \\
&= \frac{1}{1+e^{-x}}(\frac{e^{-x}+1-1}{1+e^{-x}}) = \frac{1}{1+e^{-x}}[\frac{(1+e^{-x})-1}{1+e^{-x}}] \\
&= \frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}}) \\
&= \sigma(x)[1-\sigma(x)]
\end{aligned} \tag{2}$$

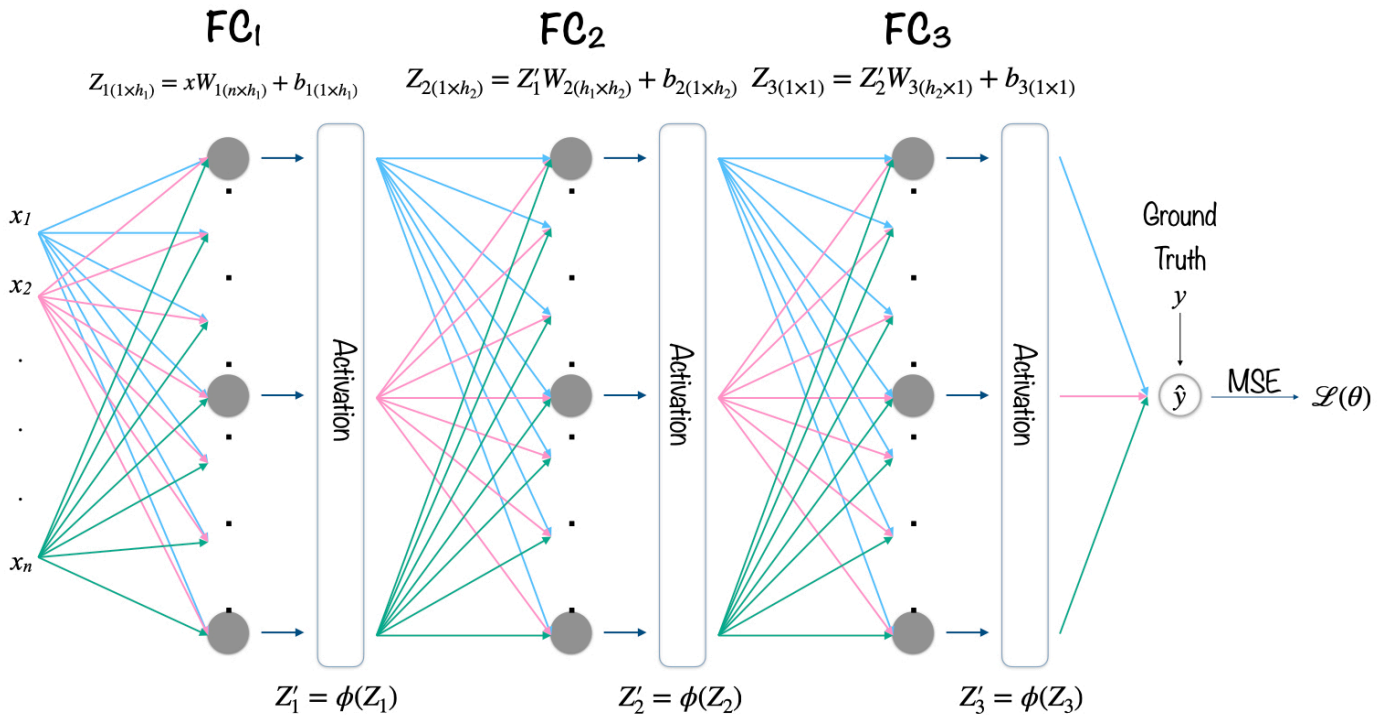
Visualizing the sigmoid function with MATLAB.




```
class Sigmoid(Layer):
    def forward(self, x):
        self.output = 1 / (1 + np.exp(-x))
        return self.output

    def backward(self, x):
        return x * (self.output * (1 - self.output))
```

Neural Network



The above figure illustrates the structure and the forward pass of a 3-layered fully connected network. When initializing, the weight matrices W_1, W_2, W_3 and biases b_1, b_2, b_3 of the three layers are small random numbers. For an input x of shape $n \times 1$, the first layer multiples it with its weight matrix W_1 and add the bias b_1, formulated as $Z_1 = xW_1 + b_1$, then use an activation function to add non-linearity to the results, $Z_1' = \phi(Z_1)$. This step is crucial when training to identify complex patterns. The second and third layer perform the same calculation using their own weight matrices and biases. At the output of the last fully connected layer, we compare the results of the prediction with the ground truth to compute the mean square error as loss to update the weight matrices and biases in the back propagation. The full calculation can be listed as:

$$
\begin{aligned}
Z_1 &= xW_1 + b_1 \\
Z_1' &= \phi(Z_1) \\
Z_2 &= xW_2 + b_2 \\
Z_2' &= \phi(Z_2) \\
Z_3 &= xW_3 + b_3 \\
Z_3' &= \phi(Z_3) \\
\mathcal{L} &= (Z_3' - y)^2
\end{aligned}
\tag{3}
$$

The implementation can be modulized into a `FullyConnectedLayer` class then utilizing the fully connected layer to enhance the readability of the code.

```
class Layer:
    def forward(self, input):
        raise NotImplementedError

    def backward(self, input_grad):
```

```
        raise NotImplementedError

class FullyConnectedLayer(Layer):
    def __init__(self, input_shape, output_shape):
        self.W = np.random.randn(input_shape, output_shape) * 0.5
        self.b = np.zeros((1, output_shape))
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)

    def forward(self, input):
        self.input = input
        return np.dot(input, self.W) + self.b

    def backward(self, output_grad):
        self.dW = np.dot(self.input.T, output_grad)
        self.db = np.sum(output_grad, axis=0, keepdims=True)
        return np.dot(output_grad, self.W.T)

    def update(self, learning_rate):
        self.W -= learning_rate * self.dW
        self.b -= learning_rate * self.db
```

With the `FullyConnectedLayer` and `Sigmoid` classes, we can implementation the entire 3-layered network by stacking the fully connected layers and perform forward pass and back propagation layer by layer.

```
class Network:
    def __init__(self, input_shape, hidden_size1, hidden_size2,
output_shape):
        self.fc1 = FullyConnectedLayer(input_shape, hidden_size1)
        self.act1 = Sigmoid()
        self.fc2 = FullyConnectedLayer(hidden_size1, hidden_size2)
        self.act2 = Sigmoid()
        self.fc3 = FullyConnectedLayer(hidden_size2, output_shape)
        self.act3 = Sigmoid()


    def forward(self, x):
        x = self.fc1.forward(x)
        x = self.act1.forward(x)
```

```
        x = self.fc2.forward(x)
        x = self.act2.forward(x)
        x = self.fc3.forward(x)
        x = self.act3.forward(x)
        return x

    def backward(self, Y):
        grad = self.act3.output - Y
        grad = self.act3.backward(grad)
        grad = self.fc3.backward(grad)
        grad = self.act2.backward(grad)
        grad = self.fc2.backward(grad)
        grad = self.act1.backward(grad)
        grad = self.fc1.backward(grad)
        return grad

    def update(self, learning_rate):
        self.fc1.update(learning_rate)
        self.fc2.update(learning_rate)
        self.fc3.update(learning_rate)
```

Back Propagation

The back propagation updates the weights and biases in the direction that minimize the loss function. By the Extreme Value theorem, the local minimum happens when the first-order deritative is 0. The gradient of each layer can be acquired using chain rule.

$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dZ_3'} \cdot \frac{dZ_3'}{dZ_3} \cdot \frac{dZ_3}{dZ_2'} \cdot \frac{dZ_2}{dZ_1'} \cdot \frac{Z_1'}{dZ_1} \cdot \frac{dZ_1}{dx} \tag{4}$$

Within each layer, we update the weights and biases by differentating the loss against the weights and bias. The $\frac{d\mathcal{L}}{dW_i}$ and $\frac{d\mathcal{L}}{db_i}$ are acquired through chain rule in the above equation.

$$\begin{aligned} W_i &\leftarrow W_i - \alpha \frac{d\mathcal{L}}{dW_i} \\ b_i &\leftarrow b_i - \alpha \frac{d\mathcal{L}}{db_i} \end{aligned} \tag{5}$$

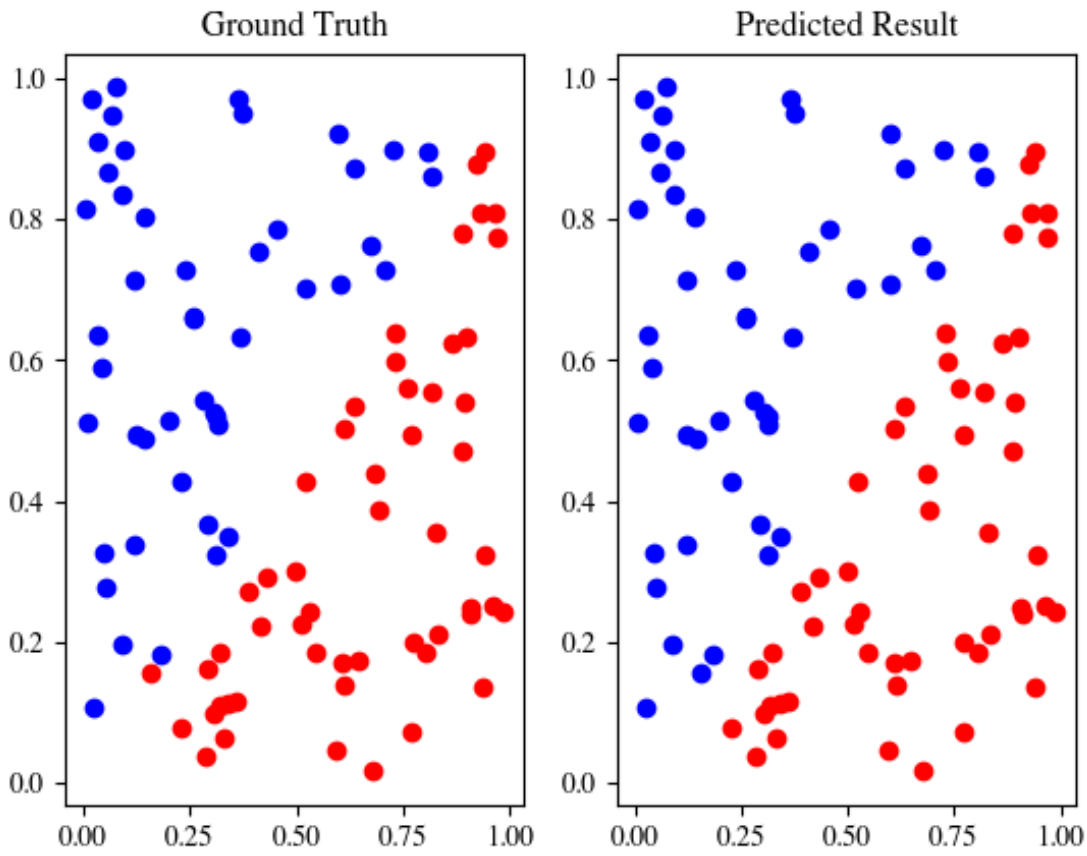
where α is the learning rate.

Through iterating between forward propagation and backpropagation, the network learns by adjusting its weights and biases to minimize the loss function, ideally improving its accuracy and ability to generalize from the training data to unseen data.

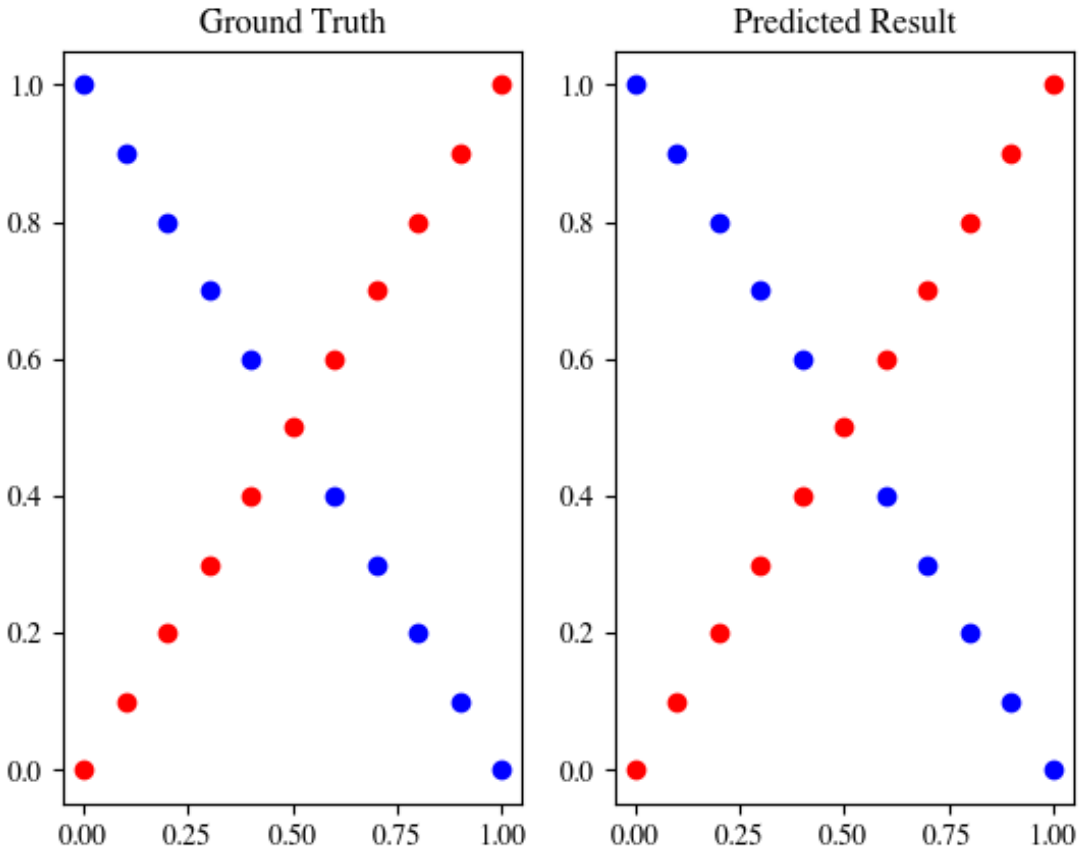
Results

Comparison Figure

- Linear Classification



- XOR



Accuracy

Linear Classification

```
Iteration: 0, Loss: 0.2505, Acc:0.4700
Iteration: 100, Loss: 0.2363, Acc:0.6200
Iteration: 200, Loss: 0.2117, Acc:0.8700
Iteration: 300, Loss: 0.1542, Acc:0.9300
Iteration: 400, Loss: 0.0954, Acc:0.9600
Iteration: 500, Loss: 0.0636, Acc:0.9700
Iteration: 600, Loss: 0.0470, Acc:0.9800
Iteration: 700, Loss: 0.0374, Acc:0.9900
Iteration: 800, Loss: 0.0313, Acc:0.9900
Iteration: 900, Loss: 0.0271, Acc:0.9900
Iteration: 1000, Loss: 0.0240, Acc:0.9900
Iteration: 1100, Loss: 0.0217, Acc:0.9900
Iteration: 1200, Loss: 0.0199, Acc:0.9900
Iteration: 1300, Loss: 0.0185, Acc:0.9900
Iteration: 1400, Loss: 0.0173, Acc:0.9900
Iteration: 1500, Loss: 0.0163, Acc:0.9900
Iteration: 1600, Loss: 0.0155, Acc:0.9900
Iteration: 1700, Loss: 0.0148, Acc:0.9900
Iteration: 1800, Loss: 0.0142, Acc:0.9900
Iteration: 1900, Loss: 0.0136, Acc:0.9900
Iteration: 2000, Loss: 0.0132, Acc:0.9900
Iteration: 2100, Loss: 0.0128, Acc:0.9900
Iteration: 2200, Loss: 0.0124, Acc:0.9900
Iteration: 2300, Loss: 0.0121, Acc:0.9900
Iteration: 2400, Loss: 0.0118, Acc:0.9900
Iteration: 2500, Loss: 0.0115, Acc:0.9900
Iteration: 2600, Loss: 0.0113, Acc:0.9900
Iteration: 2700, Loss: 0.0111, Acc:0.9900
Iteration: 2800, Loss: 0.0109, Acc:0.9900
Iteration: 2900, Loss: 0.0107, Acc:0.9900
Iteration: 3000, Loss: 0.0105, Acc:0.9900
Iteration: 3100, Loss: 0.0104, Acc:0.9900
Iteration: 3200, Loss: 0.0102, Acc:0.9900
Iteration: 3300, Loss: 0.0101, Acc:0.9900
Iteration: 3400, Loss: 0.0100, Acc:0.9900
Iteration: 3500, Loss: 0.0099, Acc:0.9900
Iteration: 3600, Loss: 0.0098, Acc:0.9900
Iteration: 3700, Loss: 0.0097, Acc:0.9900
Iteration: 3800, Loss: 0.0096, Acc:0.9900
Iteration: 3900, Loss: 0.0095, Acc:0.9900
Iteration: 4000, Loss: 0.0094, Acc:0.9900
Iteration: 4100, Loss: 0.0093, Acc:0.9900
Iteration: 4200, Loss: 0.0092, Acc:0.9900
Iteration: 4300, Loss: 0.0091, Acc:0.9900
Iteration: 4400, Loss: 0.0091, Acc:0.9900
Iteration: 4500, Loss: 0.0090, Acc:0.9900
Iteration: 4600, Loss: 0.0089, Acc:0.9900
Iteration: 4700, Loss: 0.0089, Acc:0.9900
Iteration: 4800, Loss: 0.0088, Acc:0.9900
Iteration: 4900, Loss: 0.0088, Acc:0.9900
Precision: 99.0000%
[9.99725019e-01 2.05581578e-02 6.74572384e-01 9.99761770e-01
```

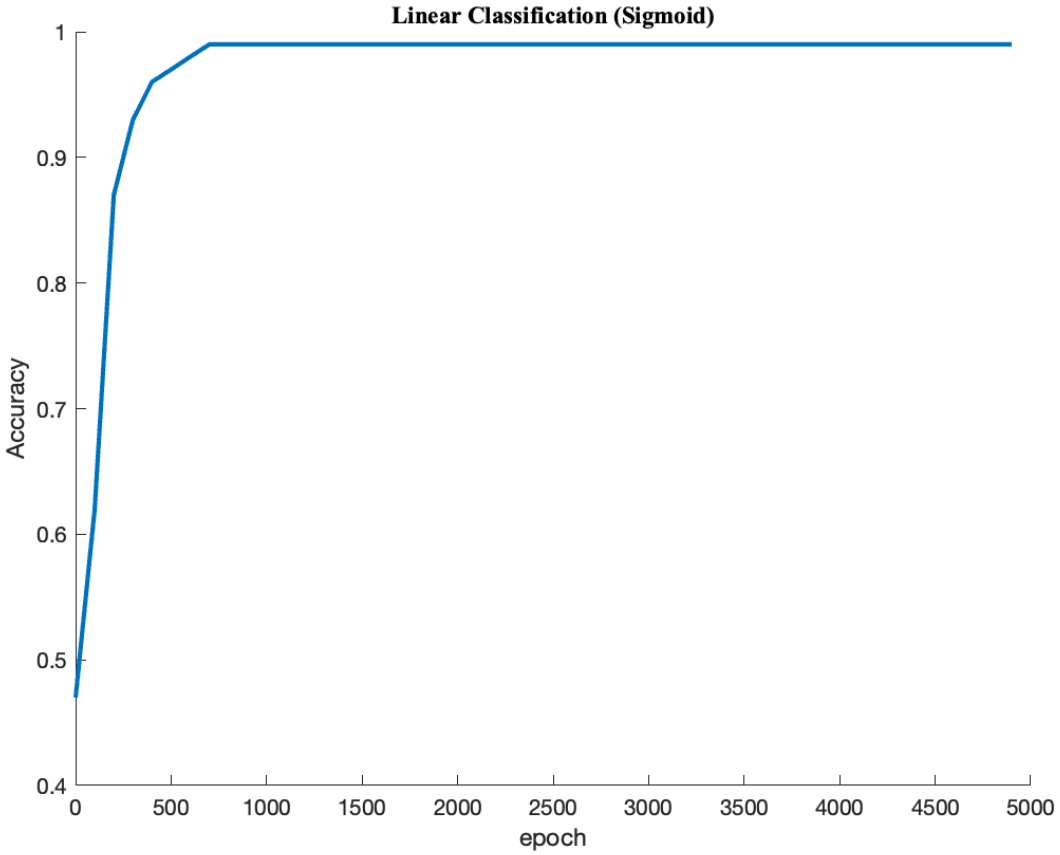
```
 9.80918897e-01 9.99768284e-01 4.86385122e-04 6.84165473e-01
 9.98315672e-01 1.96463470e-02 6.06292602e-04 9.58667576e-01
 9.99409483e-01 9.99370115e-01 5.30029607e-04 6.68008986e-04
 9.99765808e-01 1.06233535e-02 5.12434491e-03 2.56986910e-03
 9.99554279e-01 9.99765454e-01 9.99603846e-01 9.97983601e-01
 9.24598185e-04 5.09038466e-03 2.11979674e-01 9.99371106e-01
 9.84677062e-01 9.99178010e-01 3.68288269e-02 5.99430248e-04
 9.98988526e-01 9.99746115e-01 9.99766951e-01 5.08924951e-04
 9.99761977e-01 7.65980929e-01 4.63972163e-04 2.88037930e-03
 2.60413898e-03 2.12131227e-03 7.59682179e-01 6.26078363e-02
 7.04209417e-04 9.99732194e-01 4.96733109e-03 1.71331627e-03
 6.37306985e-02 9.70108853e-01 9.99735453e-01 9.97470784e-01
 4.73048211e-04 9.99469499e-01 1.65377184e-02 2.87168044e-02
 2.52809055e-02 9.98535513e-01 4.87835189e-04 9.31719839e-04
 9.63722968e-01 5.00833896e-03 9.97714552e-01 8.59057791e-01
 9.99698745e-01 6.04804346e-03 9.98304886e-01 4.85635716e-04
 9.96784640e-01 9.99733991e-01 4.60208521e-04 5.75275065e-03
 2.74409991e-03 4.39053964e-02 9.98521411e-01 4.70147728e-04
 9.99482832e-01 4.54270626e-04 9.67664902e-01 9.99688800e-01
 9.99001358e-01 5.59931252e-02 9.99756733e-01 2.43830588e-02
 9.99719140e-01 4.74006420e-04 1.66081502e-03 6.07241804e-04
 1.34971597e-03 4.46896466e-04 3.60371909e-03 1.98941664e-01
 9.99601337e-01 2.00051526e-03 1.61860252e-03 9.99761439e-01
 1.85577252e-03 7.30190492e-01 9.95459974e-01 3.77995861e-02]
```

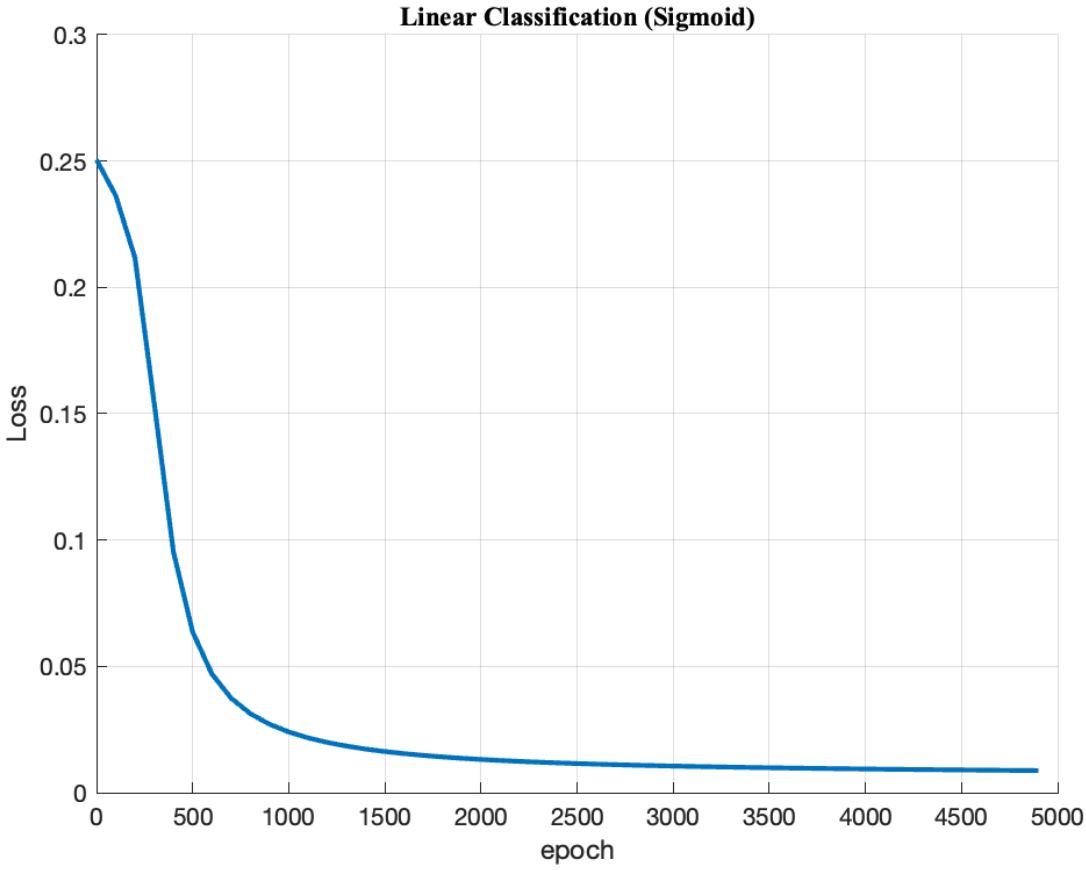
XOR

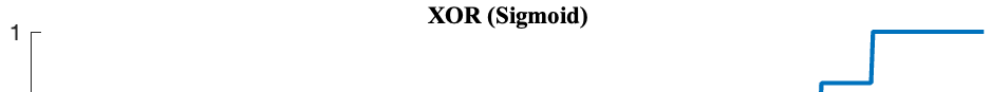
```
Iteration: 0, Loss: 0.4297, Acc:0.4762
Iteration: 100, Loss: 0.2485, Acc:0.6190
Iteration: 200, Loss: 0.2480, Acc:0.7143
Iteration: 300, Loss: 0.2476, Acc:0.6667
Iteration: 400, Loss: 0.2472, Acc:0.6667
Iteration: 500, Loss: 0.2468, Acc:0.6667
Iteration: 600, Loss: 0.2464, Acc:0.6667
Iteration: 700, Loss: 0.2460, Acc:0.6667
Iteration: 800, Loss: 0.2455, Acc:0.6667
Iteration: 900, Loss: 0.2451, Acc:0.6667
Iteration: 1000, Loss: 0.2446, Acc:0.6667
Iteration: 1100, Loss: 0.2442, Acc:0.6667
Iteration: 1200, Loss: 0.2437, Acc:0.6667
Iteration: 1300, Loss: 0.2433, Acc:0.6667
Iteration: 1400, Loss: 0.2428, Acc:0.6667
Iteration: 1500, Loss: 0.2424, Acc:0.6667
Iteration: 1600, Loss: 0.2420, Acc:0.6667
Iteration: 1700, Loss: 0.2416, Acc:0.6667
Iteration: 1800, Loss: 0.2413, Acc:0.6667
Iteration: 1900, Loss: 0.2409, Acc:0.6667
Iteration: 2000, Loss: 0.2406, Acc:0.6667
Iteration: 2100, Loss: 0.2402, Acc:0.6667
Iteration: 2200, Loss: 0.2399, Acc:0.6667
Iteration: 2300, Loss: 0.2396, Acc:0.6667
Iteration: 2400, Loss: 0.2392, Acc:0.6667
Iteration: 2500, Loss: 0.2389, Acc:0.6667
Iteration: 2600, Loss: 0.2386, Acc:0.6667
```

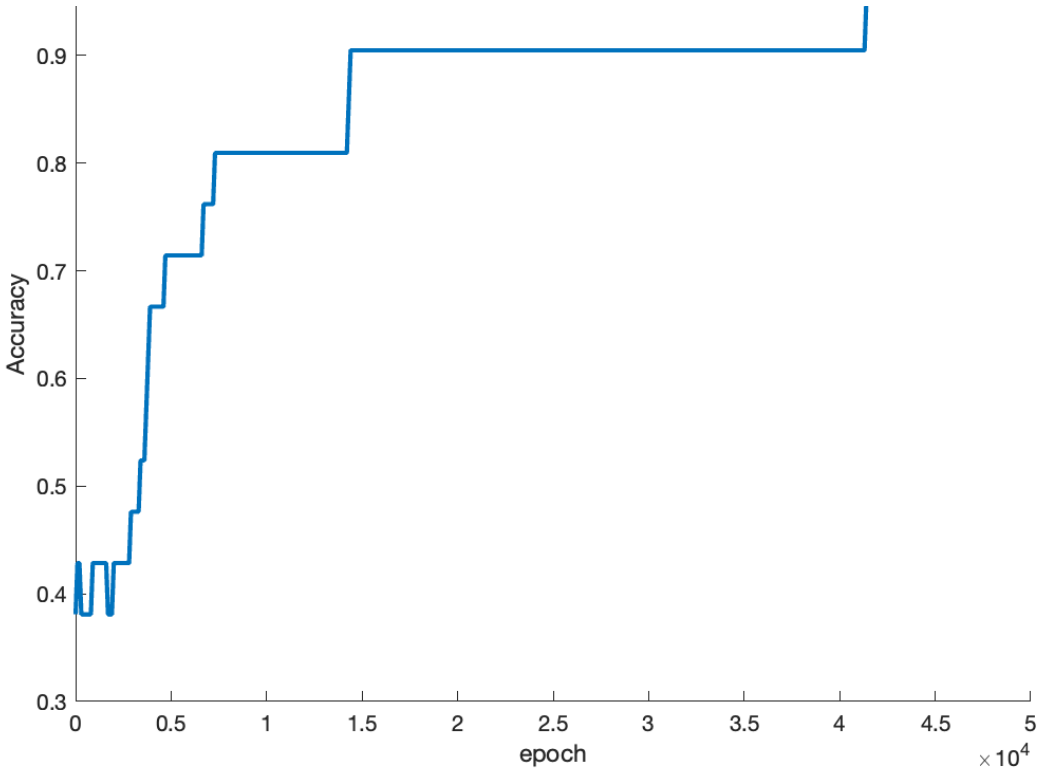
```
Iteration: 2700, Loss: 0.2383, Acc:0.6667
Iteration: 2800, Loss: 0.2380, Acc:0.6667
Iteration: 2900, Loss: 0.2377, Acc:0.6667
Iteration: 3000, Loss: 0.2373, Acc:0.6667
Iteration: 3100, Loss: 0.2370, Acc:0.6667
Iteration: 3200, Loss: 0.2367, Acc:0.6667
Iteration: 3300, Loss: 0.2364, Acc:0.6667
Iteration: 3400, Loss: 0.2361, Acc:0.6190
Iteration: 3500, Loss: 0.2357, Acc:0.6190
Iteration: 3600, Loss: 0.2354, Acc:0.6190
Iteration: 3700, Loss: 0.2351, Acc:0.6190
Iteration: 3800, Loss: 0.2347, Acc:0.6190
Iteration: 3900, Loss: 0.2344, Acc:0.6190
Iteration: 4000, Loss: 0.2341, Acc:0.6667
Iteration: 4100, Loss: 0.2337, Acc:0.6667
Iteration: 4200, Loss: 0.2334, Acc:0.6667
Iteration: 4300, Loss: 0.2330, Acc:0.6667
Iteration: 4400, Loss: 0.2327, Acc:0.6667
Iteration: 4500, Loss: 0.2323, Acc:0.6667
Iteration: 4600, Loss: 0.2319, Acc:0.6667
Iteration: 4700, Loss: 0.2316, Acc:0.6667
Iteration: 4800, Loss: 0.2312, Acc:0.7143
Iteration: 4900, Loss: 0.2308, Acc:0.7143
Iteration: 5000, Loss: 0.2304, Acc:0.7619
Iteration: 5100, Loss: 0.2301, Acc:0.7619
Iteration: 5200, Loss: 0.2297, Acc:0.7619
Iteration: 5300, Loss: 0.2293, Acc:0.7619
Iteration: 5400, Loss: 0.2289, Acc:0.7619
Iteration: 5500, Loss: 0.2285, Acc:0.7619
Iteration: 5600, Loss: 0.2281, Acc:0.7619
Iteration: 5700, Loss: 0.2276, Acc:0.7619
Precision: 100.0000%
[0.03711805 0.96603047 0.05155738 0.95725486 0.09060157 0.93648859
 0.17832596 0.86461576 0.28751182 0.56260466 0.32869875 0.28139468
 0.5172116  0.19350905 0.84036305 0.11638724 0.95936808 0.06762969
 0.98572344 0.04070148 0.99287675]
```

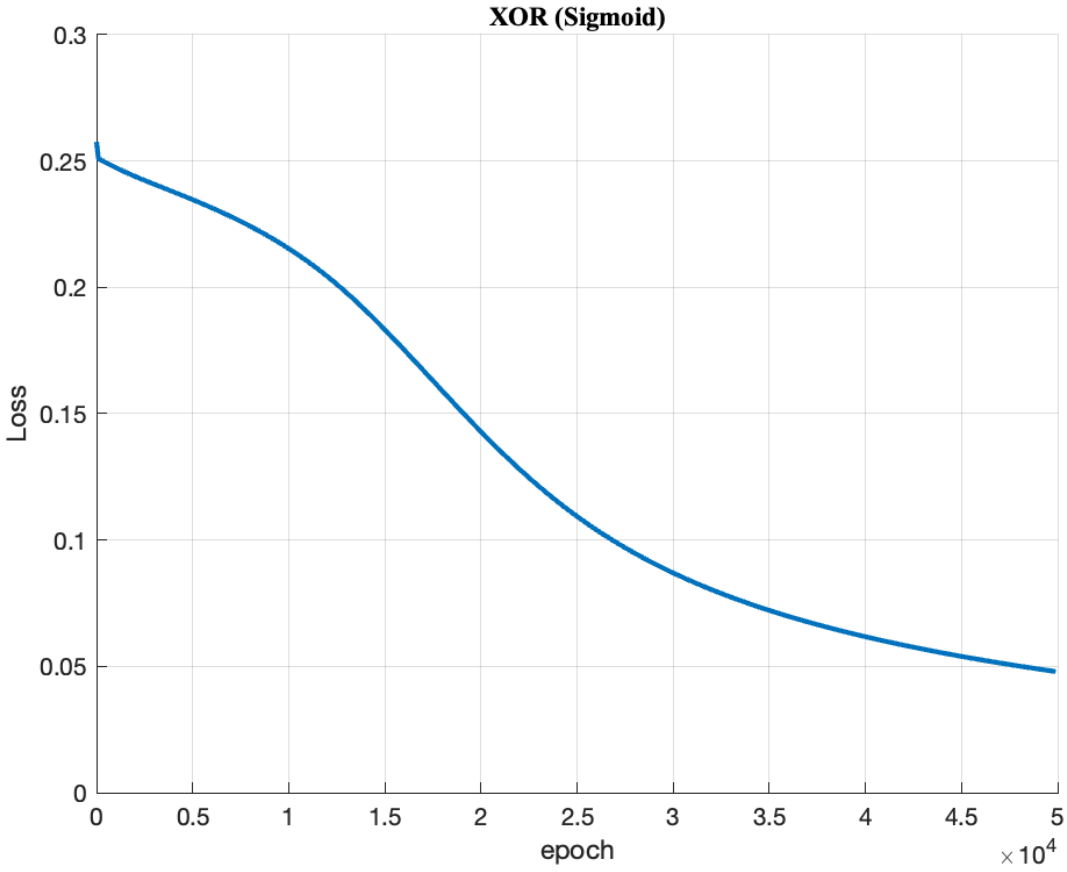
Learning Curve


Linear Classification (Sigmoid)
Accuracy
epoch
1
0.9
0.8
0.7
0.6
0.5
0.4
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


Linear Classification (Sigmoid)
Loss
epoch
0.3
0.25
0.2
0.15
0.1
0.05
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


XOR (Sigmoid)
1


Accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
epoch
×10^4


XOR (Sigmoid)
Loss
0.3
0.25
0.2
0.15
0.1
0.05
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
epoch
×10^4

Discussion

Different Learning Rates

Linear Classification (Trained for 5000 epochs)

Learning Rate	Precision
0.1	99%
0.01	99%
0.001	99%

XOR (Trained for 50000 epochs)

Learning Rate	Precision
0.1	100%
0.01	100%
0.001	100%

With enough epochs of training, the impact caused by the learning rate can be migrated.

Different Number of Hidden Units

Linear Classification (Trained for 5000 epochs)

h_1	h_2	Precision
16	4	99%
2	2	99%
128	64	99%

XOR (Trained for 50000 epochs)

h_1	h_2	Precision
128	64	100%
16	4	52.38%
2	2	52.38%

The XOR problem is without a doubt a more difficult problem, and with fewer samples, it is more difficult for the neural network to learn to predict the correct label of the point according to its coordinate, therefore, the results of less hidden units has a large impact on the perfromance on the XOR case.

The Linear Classification is a rather simple problem, therefore, the number of hidden units has minor effect on predicting the correct label.

No Activation

Linear Classification

```
Precision: 53.0000%
[nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan]
```

XOR

```
Precision: 52.3810%
[nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan]
```

Without the activation function to limit the value in the range $[0, 1]$ and add non-linearity, it is likely that the values in the weight matrices get too small or too large with large number of training epochs.

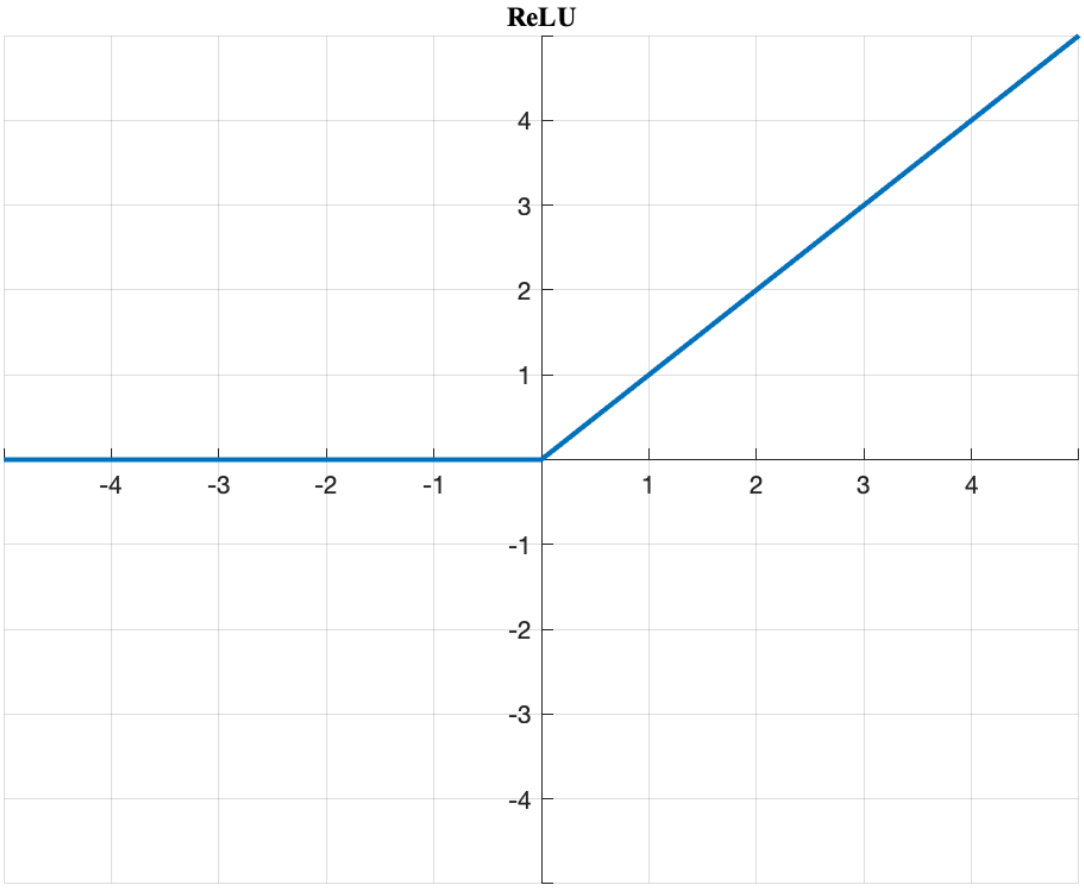
Extra

ReLU

The Recitified Linear Unit (ReLU) is a simple yet popular activation function that identifies the positive inputs and clear out the negative inputs, providing more sensitivity to the activation sum input and avoiding easy saturation as the Sigmoid activation failed to achieve. The function works by outputting the original input value if it is positive, and gives 0 when it falls below 0. The nature of ReLU function allows more representational sparsity than the Sigmoid function, where only 1 element could be 0 and behaves more like a linear function, which is easier to optimize.

$$\phi(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$\frac{d}{dx}\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$




```
class ReLU(Layer):
    def forward(self, input):
        self.input = input
        self.output = np.maximum(0, input)
        return self.output

    def backward(self, output_grad):
        input_grad = output_grad * (self.input > 0)
        return input_grad
```

- Linear Classification

```
Iteration: 0, Loss: 0.4169, Acc:0.4700
Iteration: 100, Loss: 0.4700, Acc:0.5300
Iteration: 200, Loss: 0.4700, Acc:0.5300
Iteration: 300, Loss: 0.4700, Acc:0.5300
Iteration: 400, Loss: 0.4700, Acc:0.5300
```

```
Iteration: 500, Loss: 0.4700, Acc:0.5300
Iteration: 600, Loss: 0.4700, Acc:0.5300
Iteration: 700, Loss: 0.4700, Acc:0.5300
Iteration: 800, Loss: 0.4700, Acc:0.5300
Iteration: 900, Loss: 0.4700, Acc:0.5300
Iteration: 1000, Loss: 0.4700, Acc:0.5300
Iteration: 1100, Loss: 0.4700, Acc:0.5300
Iteration: 1200, Loss: 0.4700, Acc:0.5300
Iteration: 1300, Loss: 0.4700, Acc:0.5300
Iteration: 1400, Loss: 0.4700, Acc:0.5300
Iteration: 1500, Loss: 0.4700, Acc:0.5300
Iteration: 1600, Loss: 0.4700, Acc:0.5300
Iteration: 1700, Loss: 0.4700, Acc:0.5300
Iteration: 1800, Loss: 0.4700, Acc:0.5300
Iteration: 1900, Loss: 0.4700, Acc:0.5300
Iteration: 2000, Loss: 0.4700, Acc:0.5300
Iteration: 2100, Loss: 0.4700, Acc:0.5300
Iteration: 2200, Loss: 0.4700, Acc:0.5300
Iteration: 2300, Loss: 0.4700, Acc:0.5300
Iteration: 2400, Loss: 0.4700, Acc:0.5300
Iteration: 2500, Loss: 0.4700, Acc:0.5300
Iteration: 2600, Loss: 0.4700, Acc:0.5300
Iteration: 2700, Loss: 0.4700, Acc:0.5300
Iteration: 2800, Loss: 0.4700, Acc:0.5300
Iteration: 2900, Loss: 0.4700, Acc:0.5300
Iteration: 3000, Loss: 0.4700, Acc:0.5300
Iteration: 3100, Loss: 0.4700, Acc:0.5300
Iteration: 3200, Loss: 0.4700, Acc:0.5300
Iteration: 3300, Loss: 0.4700, Acc:0.5300
Iteration: 3400, Loss: 0.4700, Acc:0.5300
Iteration: 3500, Loss: 0.4700, Acc:0.5300
Iteration: 3600, Loss: 0.4700, Acc:0.5300
Iteration: 3700, Loss: 0.4700, Acc:0.5300
Iteration: 3800, Loss: 0.4700, Acc:0.5300
Iteration: 3900, Loss: 0.4700, Acc:0.5300
Iteration: 4000, Loss: 0.4700, Acc:0.5300
Iteration: 4100, Loss: 0.4700, Acc:0.5300
Iteration: 4200, Loss: 0.4700, Acc:0.5300
Iteration: 4300, Loss: 0.4700, Acc:0.5300
Iteration: 4400, Loss: 0.4700, Acc:0.5300
Iteration: 4500, Loss: 0.4700, Acc:0.5300
Iteration: 4600, Loss: 0.4700, Acc:0.5300
Iteration: 4700, Loss: 0.4700, Acc:0.5300
Iteration: 4800, Loss: 0.4700, Acc:0.5300
Iteration: 4900, Loss: 0.4700, Acc:0.5300
Precision: 53.0000%
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
```

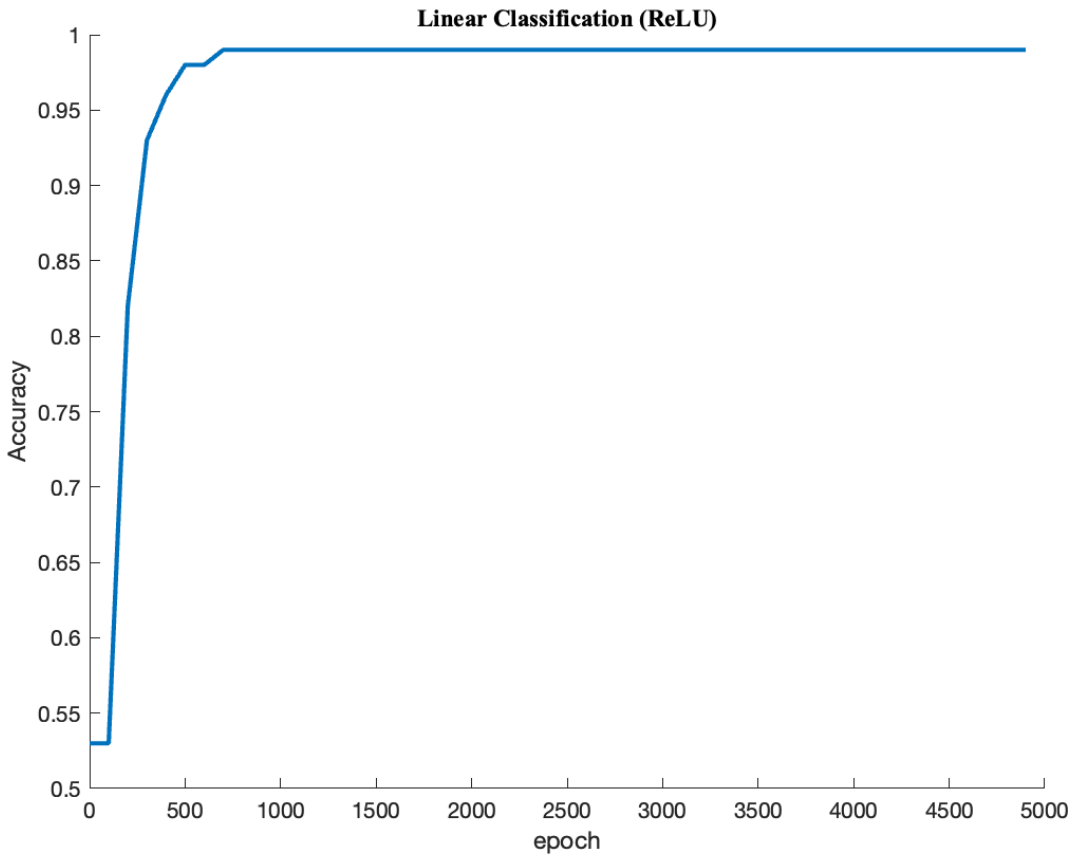
- XOR

```
Precision: 52.3810%
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

By adjusting the learning rate to 0.001 on both models and lower the values in weight initialization, we also obtain high precision models.

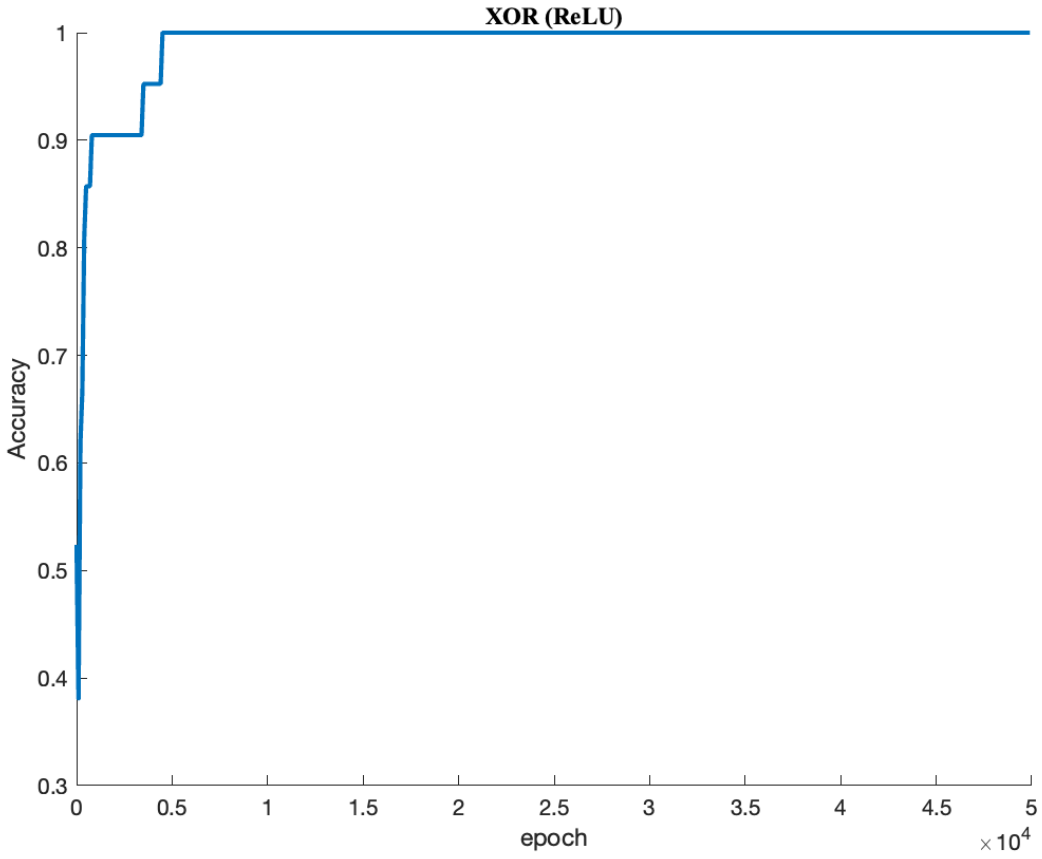
- Linear Classification

```
Precision: 99.0000%
[1.00090586 0.         0.69426748 1.0037425  0.99680147 1.0048566
 0.         0.70372233 0.99875122 0.         0.         0.99779847
 0.99889394 0.99980763 0.         0.         1.00423298 0.
 0.         0.         1.00051504 1.00429326 1.000184   0.99863856
 0.         0.         0.16640017 0.99829839 0.99883078 1.00018228
 0.         0.         0.99912542 1.00241102 1.00439038 0.
 1.00396283 0.79081622 0.         0.         0.         0.
 0.79116502 0.         0.         1.00203283 0.         0.
 0.         0.99890891 1.00245871 0.99853138 0.         0.99918433
 0.         0.         0.         0.99757049 0.         0.
 0.99586096 0.         0.99890679 0.9507951  1.0018634  0.
 0.99946089 0.         0.99765525 1.00116768 0.         0.
 0.         0.         0.99979554 0.         1.00022925 0.
 0.99640132 1.00086144 0.99880111 0.         1.00318382 0.
 1.00204656 0.         0.         0.         0.         0.
 0.         0.14558604 1.00017569 0.         0.         1.00357691
 0.         0.75111348 0.99674721 0.        ]
```

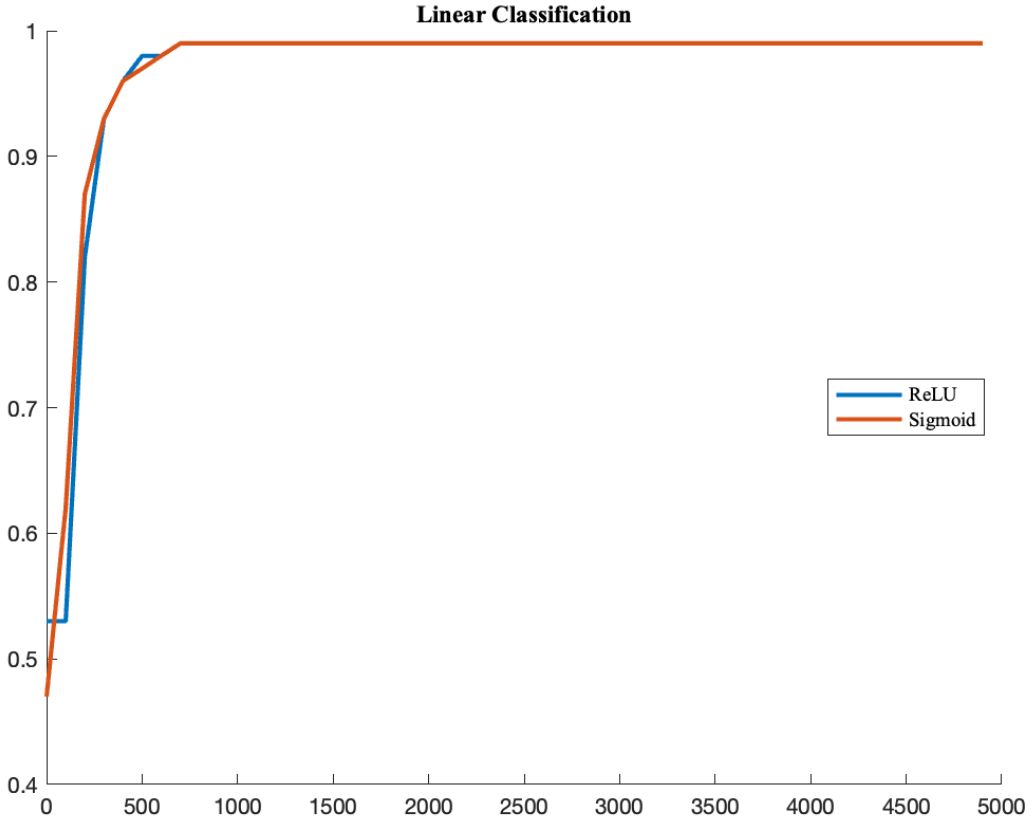


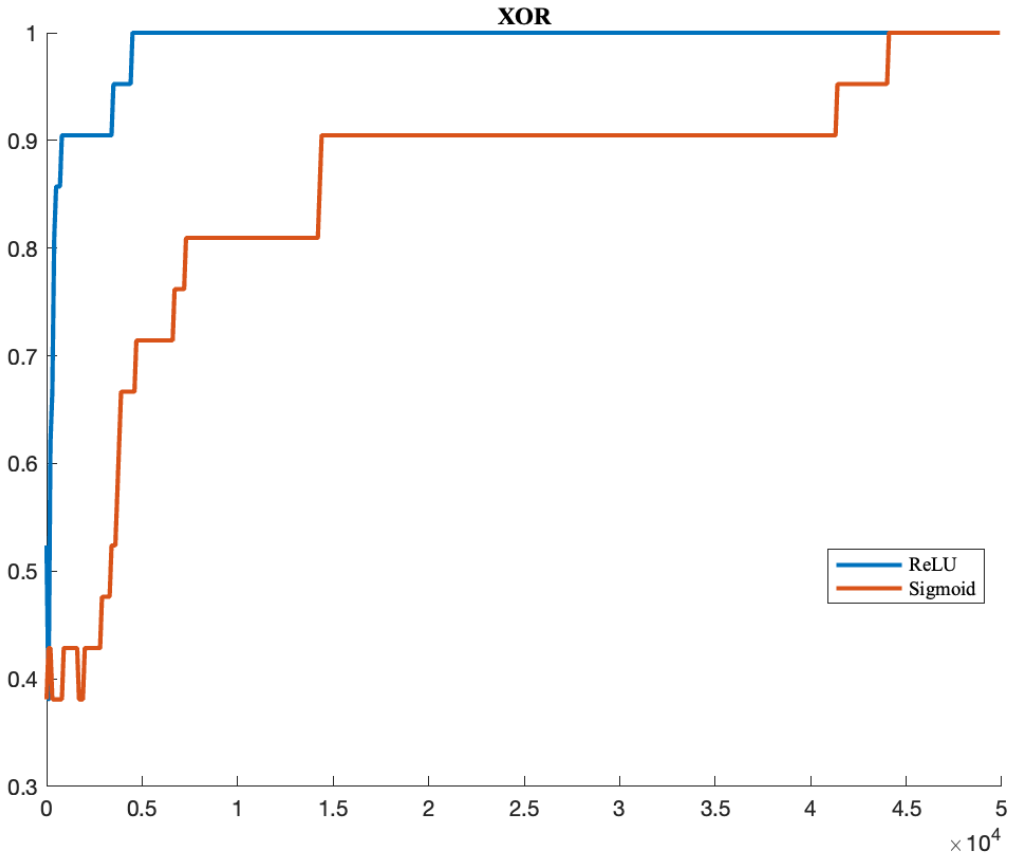
- XOR

```
Precision: 100.0000%
[0.00000000e+00 1.00009033e+00 0.00000000e+00 9.99749417e-01
 0.00000000e+00 1.00018708e+00 0.00000000e+00 1.00062475e+00
 0.00000000e+00 9.98556971e-01 9.35309169e-04 0.00000000e+00
 9.98673517e-01 0.00000000e+00 1.00190908e+00 0.00000000e+00
 1.00037084e+00 0.00000000e+00 9.98832609e-01 0.00000000e+00
 1.00016742e+00]
```



The learning curve comparisons between using `ReLU` or `Sigmoid` as the activation function are shown as follows:


Linear Classification
1
0.9
0.8
0.7
0.6
0.5
0.4
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
ReLU
Sigmoid


XOR
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
×10^4
ReLU
Sigmoid

Using the `ReLU` as the activation function makes no obvious difference in the Linear Classification problem, but yields significantly faster convergence in the XOR problem. The reason behind this phenomenon is because Linear Classification is a relative simple problem, even with small weight matrices and few neurons could achieve a high accuracy prediction, but the XOR problem is more complicated and thus requires more neurons to reach higher accuracy predictions.

Alison Wen