Lab1 Report

0856091 蘇邱弘

July 2020

1 Introduction

Backpropagation is a method used to calculate the gradient and to update the weights in most of the deep learning's neural networks. In this lab, I implement a simple 2-layer fully connected neural network, which using backpropagation to update it's weight during the training process, to perform the binary classification task with two different data sets: linear data and XOR data.
In this report, I will first introduce the experiment setups, including the activation function in my model, the construction process of the neural network, and the implementation detail of the backpropagation. Next, I will show my testing results on the two data sets. Finally, I will do some discussions.

2 Experiment Setups

2.1 Sigmoid Function

In this lab, sigmoid function is used as the activation function at the end of each hidden layers and the output layer in the neural network to obtain the output. In addition, the derivative of sigmoid function is used to calculate the gradient of weights. Sigmoid function and it's derivative are implemented as shown in Figure 1.

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

def derivative_sigmoid(self, x):
    s = self.sigmoid(x)
    return np.multiply(s, 1.0 - s)
```

Figure 1: Implementation of sigmoid function and it's derivative

2.2 Neural Network

The neural network of this lab is constructed as follows:
First of all, it is a fully connected network taking two input and one output, and there are

two hidden layers in total. The width of each hidden layer can be determined by passing the parameters to the constructor of the network. Furthermore, we can decide that whether to use sigmoid function as the model's activation function or not.
After deciding the overall structure, for each weight in the network, it will be assigned a initial number, which is randomly sampled from the standard normal distribution.
The overall construction process of the network is implemented as shown in Figure 2.

```python
class FC_Net:
    def __init__(self, first_hidden_width, second_hidden_width, with_sigmoid=True):
        input_width = 2
        output_width = 1
        self.with_sigmoid = with_sigmoid

        # random generates weights
        self.W = []
        self.W.append(np.random.randn(input_width, first_hidden_width))
        self.W.append(np.random.randn(first_hidden_width, second_hidden_width))
        self.W.append(np.random.randn(second_hidden_width, output_width))
```

Figure 2: Implementation of the construction process of the network

2.3 Backpropagation

For the training process in each epoch, I mainly divide the backpropagation into two parts: forward pass and backward pass. After updating the weights in the network, I use mean square error as the cost function and the overall accuracy compared to the ground truth to estimate the output result. The training process is implemented as shown in Figure 3. Next, I will introduce the main procedure of forward pass and backward pass.

2.3.1 Forward Pass

During the forward pass, the vector before and after multiplying each weight matrix in the network are derived by forwarding the input to the output layer through the network. The formulation of the forward pass is as follows:
(Note that the weight matrix in the formulation is counted from zero.)

- Vector before multiplying the i-th weight matrix:

$$a^i = \begin{cases} \text{input of the network}, & \text{if } i = 0 \\ \sigma(z^{i-1}), & \text{otherwise} \end{cases}$$

- Vector after multiplying the i-th weight matrix:

$$z^i = W^i a^i$$

Each a^i and z^i are then stored in order to perform the backward pass process. The forward pass process is implemented as shown in Figure 4.

```python
def train(self, x, y, epochs, learning_rate, early_stop=False):
    history = {'epochs':[], 'loss':[], 'accuracy':[]}
    print('Start training...')

    for epoch in range(1, epochs + 1):
        # backpropagation
        y_hat = self.forward(x)
        self.backward(y, y_hat, learning_rate)

        # calculate loss and accuracy
        loss = self.MSE(y, y_hat)
        acc = cal_accuracy(y, y_hat)

        # record the values
        history['epochs'].append(epoch)
        history['loss'].append(loss)
        history['accuracy'].append(acc)

        # print information
        if (epoch % int(epochs / 20)) == 0:
            print(f'epoch {epoch:<6} loss: {loss:.5f}  accuracy: {acc:.5f}')

        # early stopping
        if early_stop and acc == 1.0:
            print(f'epoch {epoch:<6} loss: {loss:.5f}  accuracy: {acc:.5f}')
            print('Accuracy = 1.0, early stop the training.')
            break

    return history
```

Figure 3: Implementation of the training process

```python
def forward(self, x):
    self.z = []
    self.a = [x]

    for i in range(len(self.W)):
        self.z.append(np.matmul(self.a[i], self.W[i]))
        if (not self.with_sigmoid) and i < (len(self.W) - 1):
            self.a.append(self.z[i])
        else:
            self.a.append(self.sigmoid(self.z[i]))

    return self.a[-1]
```

Figure 4: Implementation of the forward pass process

2.3.2 Backward Pass

After the forward pass, we can derive the gradient of the cost function C with respect to each weight in the backward direction (i.e., from $i = 2$ to 0) as follows:

$$\frac{\partial C}{\partial W^i} = \frac{\partial C}{\partial z^i}\frac{\partial z^i}{\partial W^i} = \delta^i {a^i}^T$$

Where δ^i is derived by:

$$\delta^i = \frac{\partial C}{\partial z^i} = \begin{cases} \sigma'(z^i) \times \frac{\partial C}{\partial \hat{y}}, & \text{if } i{=}2 \\ {W^{i+1}}^T \delta^{i+1} \times \sigma'(z^i), & \text{otherwise} \end{cases}$$

Each time we obtain the gradient, we can update the weights by $W^i = W^i - \rho \times \frac{\partial C}{\partial W^i}$, where ρ is the learning rate. The backward pass process is implemented as shown in Figure 5.

```
def backward(self, y, y_hat, learning_rate):
    deltas = [None for i in range(len(self.W))]
    deltas[-1] = self.derivative_sigmoid(self.z[-1]) * self.derivative_MSE(y, y_hat)

    for i in range(len(self.W) - 1, -1, -1):
        if (i - 1) >= 0:
            if self.with_sigmoid:
                deltas[i - 1] = np.matmul(deltas[i], self.W[i].T) * self.derivative_sigmoid(self.z[i - 1])
            else:
                deltas[i - 1] = np.matmul(deltas[i], self.W[i].T)

        # update weights
        gradient_to_w = np.matmul(self.a[i].T, deltas[i])
        self.W[i] -= learning_rate * gradient_to_w
```

Figure 5: Implementation of the backward pass process

3 Results of Testing

For the experiment setup, the number of epochs is set to 50000, and the learning rate is set to 0.1 for both linear and XOR data. For the network's architecture, I set width = 4 for both hidden layers of linear data classifier, and width = 10 for both hidden layers of XOR data classifier.

3.1 Screenshot and Comparison Figure

The screenshots of the training and testing processes are shown in Figure 6 and 7, and the comparison figures are shown in Figure 8. As shown in these figures, the accuracy for both two data sets can achieve 1.0.

```
Start training...
epoch 2500   loss: 0.05857  accuracy: 0.98000
epoch 5000   loss: 0.02878  accuracy: 1.00000
epoch 7500   loss: 0.02103  accuracy: 1.00000
epoch 10000  loss: 0.01680  accuracy: 1.00000
epoch 12500  loss: 0.01402  accuracy: 1.00000
epoch 15000  loss: 0.01203  accuracy: 1.00000
epoch 17500  loss: 0.01053  accuracy: 1.00000
epoch 20000  loss: 0.00936  accuracy: 1.00000
epoch 22500  loss: 0.00843  accuracy: 1.00000
epoch 25000  loss: 0.00766  accuracy: 1.00000
epoch 27500  loss: 0.00702  accuracy: 1.00000
epoch 30000  loss: 0.00648  accuracy: 1.00000
epoch 32500  loss: 0.00601  accuracy: 1.00000
epoch 35000  loss: 0.00561  accuracy: 1.00000
epoch 37500  loss: 0.00525  accuracy: 1.00000
epoch 40000  loss: 0.00493  accuracy: 1.00000
epoch 42500  loss: 0.00465  accuracy: 1.00000
epoch 45000  loss: 0.00440  accuracy: 1.00000
epoch 47500  loss: 0.00416  accuracy: 1.00000
epoch 50000  loss: 0.00396  accuracy: 1.00000
```

(a) Training

```
[[9.99779383e-01]
 [3.20817612e-05]
 [9.99846722e-01]
 [3.18540250e-01]
 [9.99788408e-01]
 [6.24944647e-01]
 [8.76205881e-05]
 [2.26164285e-06]
 [9.95247884e-01]
 [1.01562035e-01]
 [8.40718712e-01]
 [4.97373925e-06]
 [9.99840228e-01]
 [2.10504767e-06]
 [2.19870214e-06]
 [2.14889840e-06]
 [8.48758747e-06]
 [1.99629651e-06]
 [9.66658697e-03]
 [2.29448032e-06]
 [6.18804513e-06]
 [8.50924611e-05]
```

(b) Testing

Figure 6: Training and testing output of linear data

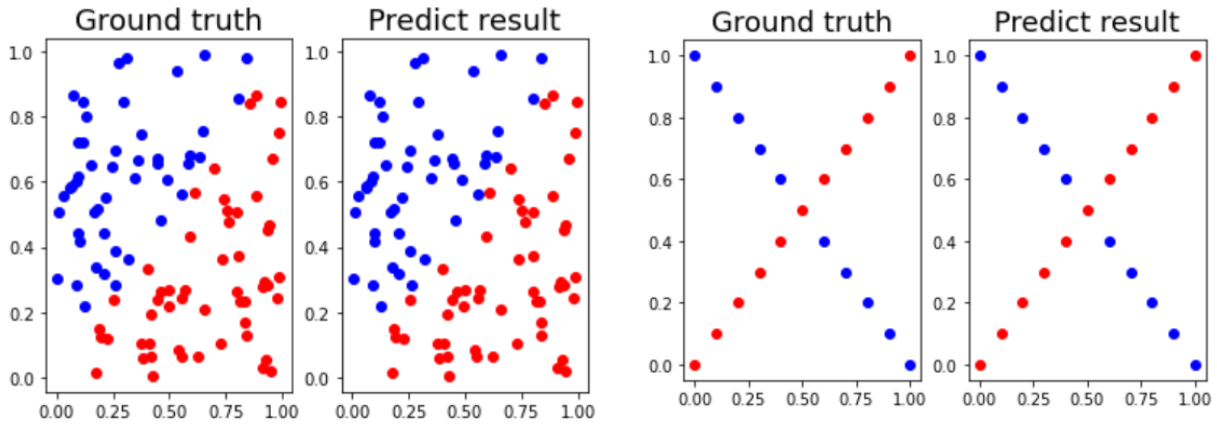
```
Start training...
epoch 2500   loss: 0.23117  accuracy: 0.71429
epoch 5000   loss: 0.10315  accuracy: 0.90476
epoch 7500   loss: 0.04426  accuracy: 0.95238
epoch 10000  loss: 0.01916  accuracy: 1.00000
epoch 12500  loss: 0.00907  accuracy: 1.00000
epoch 15000  loss: 0.00522  accuracy: 1.00000
epoch 17500  loss: 0.00346  accuracy: 1.00000
epoch 20000  loss: 0.00251  accuracy: 1.00000
epoch 22500  loss: 0.00194  accuracy: 1.00000
epoch 25000  loss: 0.00156  accuracy: 1.00000
epoch 27500  loss: 0.00130  accuracy: 1.00000
epoch 30000  loss: 0.00111  accuracy: 1.00000
epoch 32500  loss: 0.00096  accuracy: 1.00000
epoch 35000  loss: 0.00085  accuracy: 1.00000
epoch 37500  loss: 0.00075  accuracy: 1.00000
epoch 40000  loss: 0.00068  accuracy: 1.00000
epoch 42500  loss: 0.00062  accuracy: 1.00000
epoch 45000  loss: 0.00056  accuracy: 1.00000
epoch 47500  loss: 0.00052  accuracy: 1.00000
epoch 50000  loss: 0.00048  accuracy: 1.00000
```

(a) Training

```
[[0.0164866 ]
 [0.99666994]
 [0.01807389]
 [0.99635116]
 [0.01977818]
 [0.99553857]
 [0.02138652]
 [0.9927731 ]
 [0.02265626]
 [0.94873007]
 [0.02339836]
 [0.02354399]
 [0.9512588 ]
 [0.02315397]
 [0.99812142]
 [0.02237482]
 [0.99892891]
 [0.02137735]
 [0.99906083]
 [0.0203106 ]
 [0.99910017]]
```

(b) Testing

Figure 7: Training and testing output of XOR data



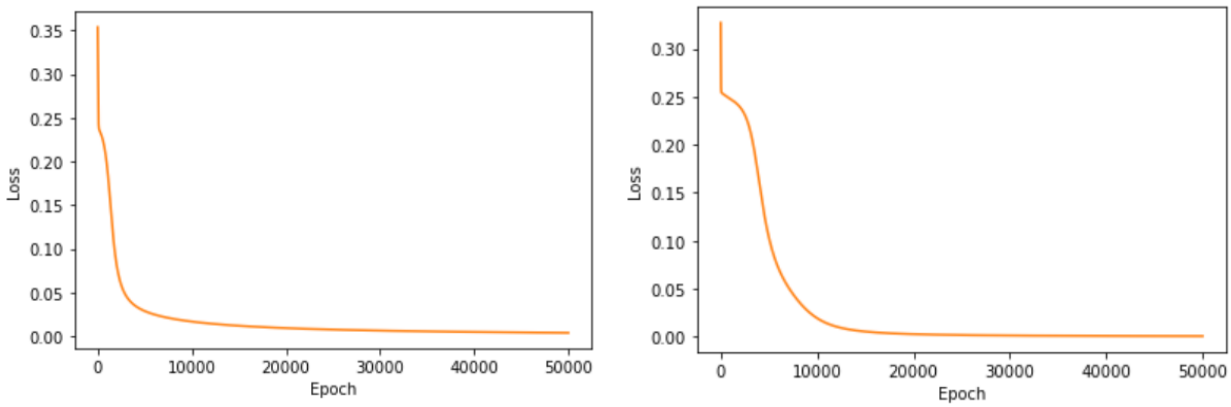
(a) Linear data

(b) XOR data

Figure 8: Comparison graph

3.2 Learning Curve

The learning curve of both data sets are shown in Figure 9. Here we can observe an interesting phenomenon that the loss drops fast in the beginning, then the loss drop become smoother when it drops to about 0.25.



(a) Linear data

(b) XOR data

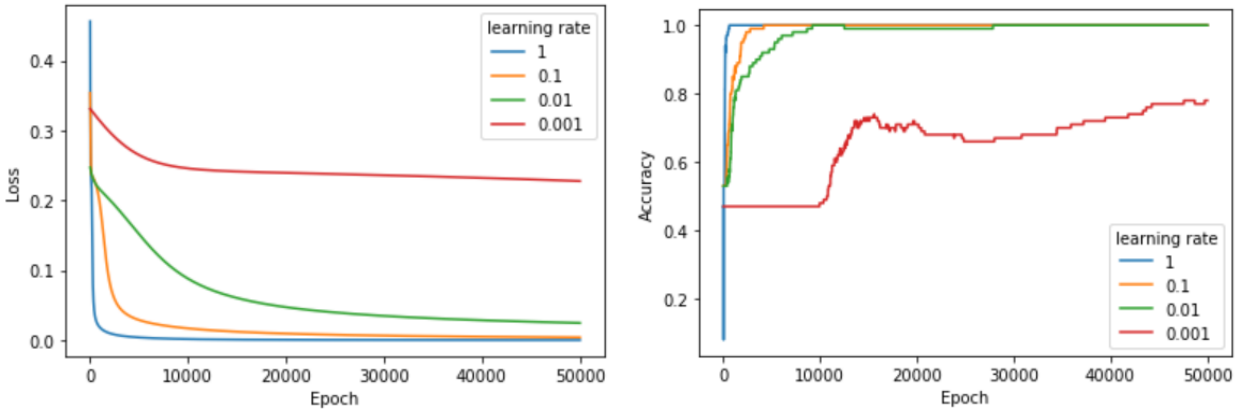
Figure 9: Learning curve

4 Discussion

4.1 Different Learning Rates

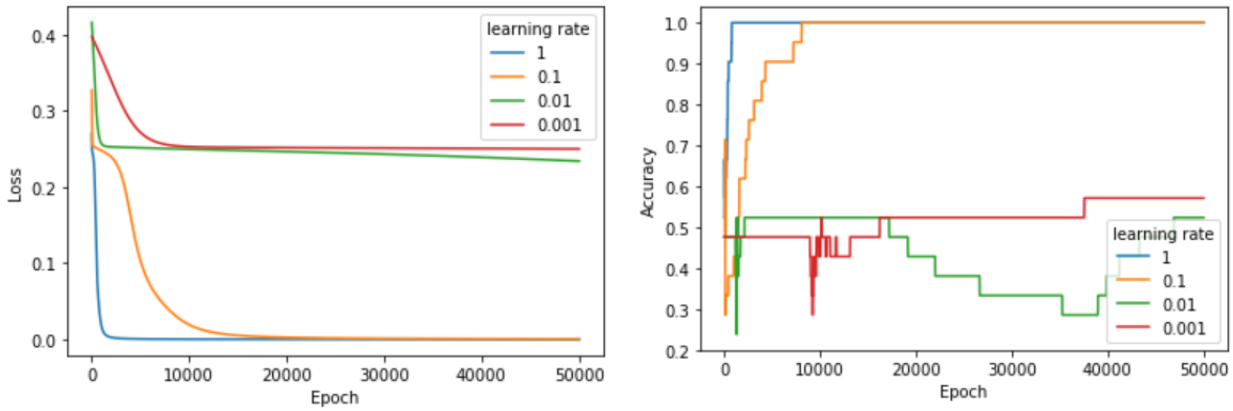
In this section, I do the experiment with different learning rates while the other settings remain the same as Section 3. The results are shown in Figure 10 and 11.
As shown in these figures, in both cases of linear data and XOR data, we can observe that

with larger learning rate, both the loss and the accuracy value converge faster, and will get a better result after training.



(a) Loss (b) Accuracy

Figure 10: Training result of linear data classifier with different learning rate

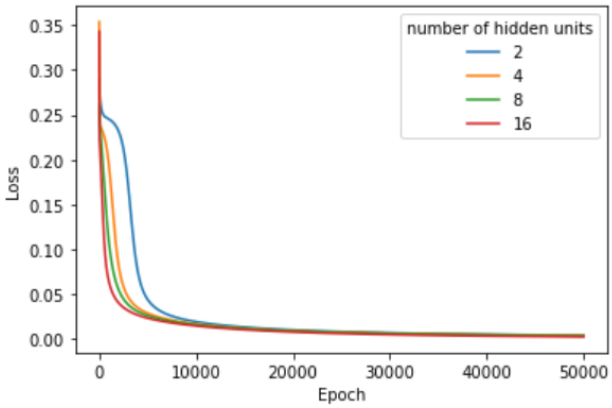


(a) Loss (b) Accuracy

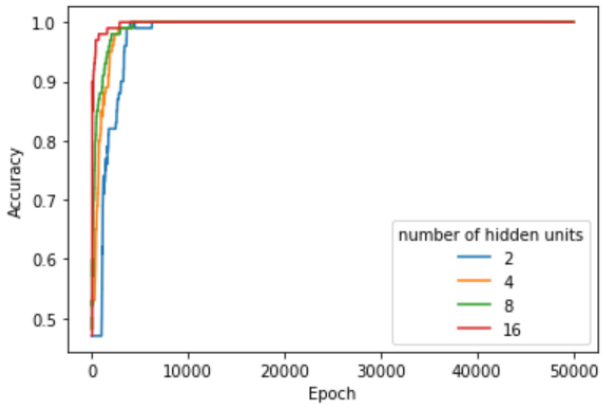
Figure 11: Training result of XOR data classifier with different learning rate

4.2 Different Numbers of Hidden Units

In this section, I do the experiment with different numbers of hidden units while the other settings remain the same as Section 3. Note that the two hidden layer's unit numbers are set to be same here. The results are shown in Figure 12 and 13.
As shown in these figures, with more units in the hidden layers, both the loss and the accuracy value converge faster, and all the cases can obtain a good result after training if the number of epochs are large enough.

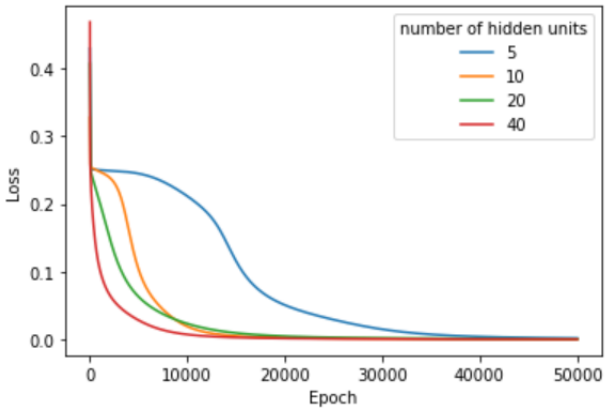


(a) Loss

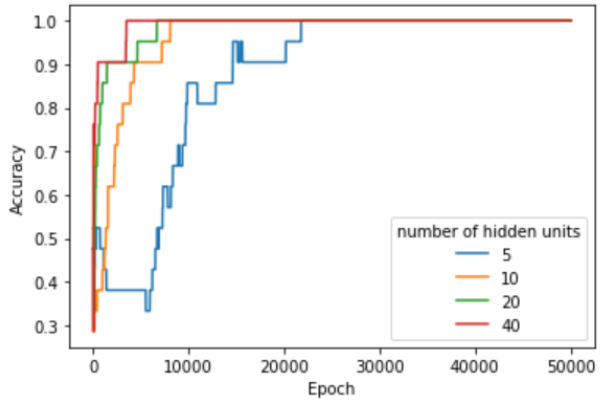


(b) Accuracy

Figure 12: Training result of linear data classifier with different numbers of hidden units



(a) Loss



(b) Accuracy

Figure 13: Training result of XOR data classifier with different numbers of hidden units

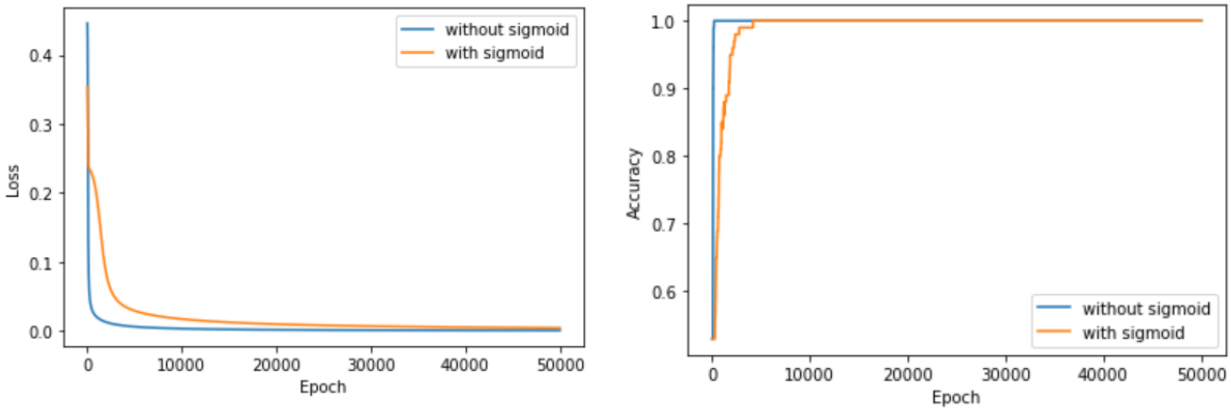
4.3 Without Sigmoid Functions

In this section, I do the experiment of replacing all the activation functions from sigmoid function to linear function in the hidden layers, while the other settings remain the same as Section 3. The results are shown in Figure 14 and 15.

As shown in these figures, we can see that with linear data, the training result without sigmoid functions converge faster than the one with sigmoid function. On the contrary, with XOR data, it seems that the classifier without sigmoid functions can't classify the data precisely.

The reason of the results may be caused by the characteristic of the ground truth classification model and the sigmoid function. For linear data, we have already known that the ground truth classification model is a linear line, so when we remove the sigmoid functions, it means that we move the nonlinearity out of our model, so it can fit the ground truth model better. However, for the case of the XOR data, the ground truth classification model is a nonlinear

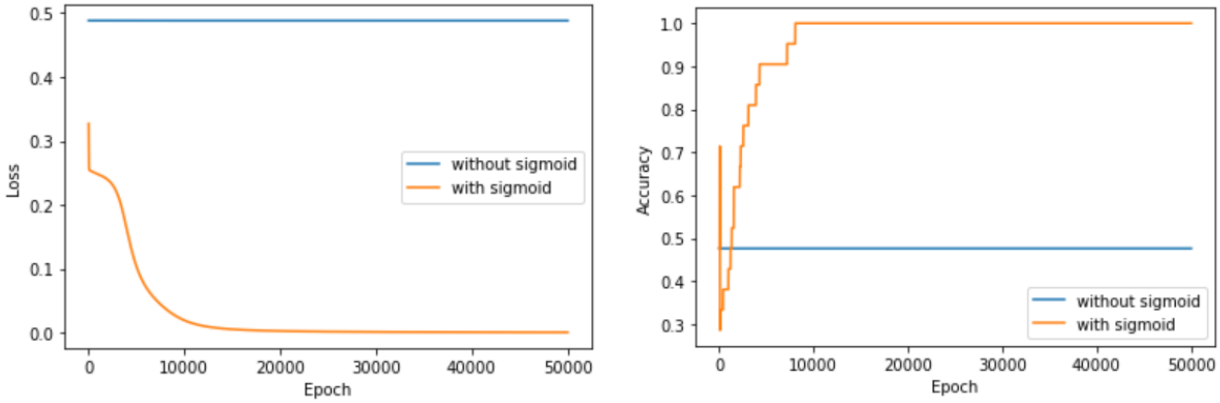
function, so if we remove the nonlinearity in our model, it may be much more difficult for it to fit this nonlinear function well.



(a) Loss

(b) Accuracy

Figure 14: Training result of linear data classifier with and without sigmoid function



(a) Loss

(b) Accuracy

Figure 15: Training result of XOR data classifier with and without sigmoid function