Lab 1 Report

Deep Learning

Name: Kai-Jie Lin
Student ID: 110652019

March 26, 2024

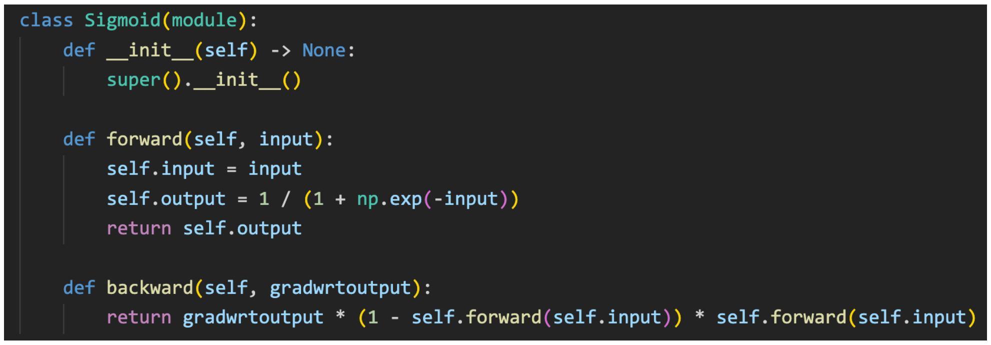
1 Introduction

In this lab, I implemented a simple neural network using the Numpy library. The neural network was trained on the linear data and XOR data. I implemented the forward and backward propagation, linear layer, various activation functions(Sigmoid, ReLU) and optimizers(SGD, Adagrad, Momentum). The result of the neural network performs differently under different configurations.

2 Experiment setups

2.1 Sigmoid functions

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



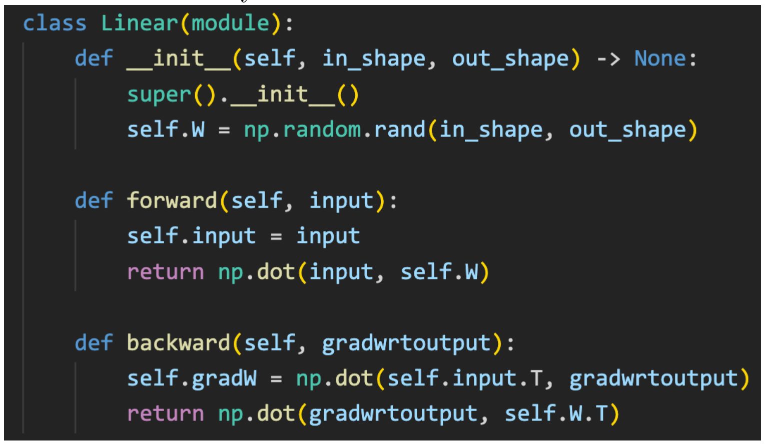
```python
class Sigmoid(module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self, input):
        self.input = input
        self.output = 1 / (1 + np.exp(-input))
        return self.output

    def backward(self, gradwrtoutput):
        return gradwrtoutput * (1 - self.forward(self.input)) * self.forward(self.input)
```

2.2 Neural network

W_1, W_2, W_3 are the random initialized weights of the neural network. σ is the sigmoid function.

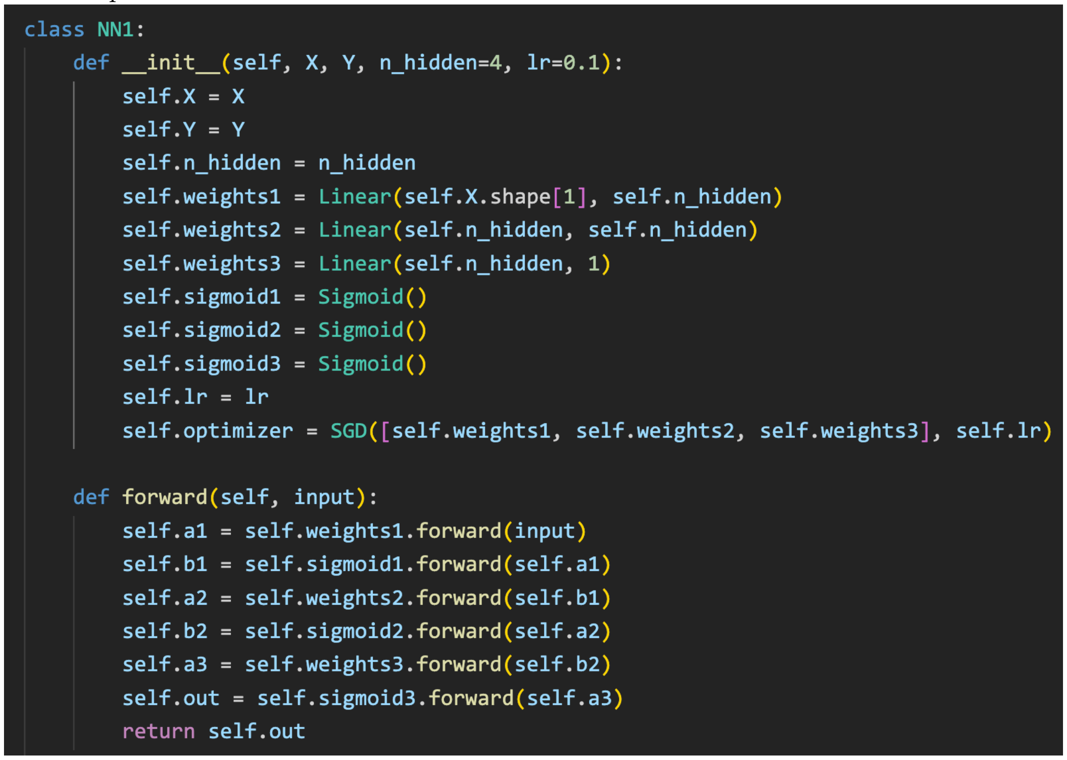
$$\begin{aligned} a_1 &= W_1 x \\ b_1 &= \sigma(a_1) \\ a_2 &= W_2 b_1 \\ b_2 &= \sigma(a_2) \\ a_3 &= W_3 b_2 \\ \hat{y} &= \sigma(a_3) \end{aligned}$$

Self defined linear layer module:



```python
class Linear(module):
    def __init__(self, in_shape, out_shape) -> None:
        super().__init__()
        self.W = np.random.rand(in_shape, out_shape)

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

    def backward(self, gradwrtoutput):
        self.gradW = np.dot(self.input.T, gradwrtoutput)
        return np.dot(gradwrtoutput, self.W.T)
```

Forward part of the neural network:



```python
class NN1:
    def __init__(self, X, Y, n_hidden=4, lr=0.1):
        self.X = X
        self.Y = Y
        self.n_hidden = n_hidden
        self.weights1 = Linear(self.X.shape[1], self.n_hidden)
        self.weights2 = Linear(self.n_hidden, self.n_hidden)
        self.weights3 = Linear(self.n_hidden, 1)
        self.sigmoid1 = Sigmoid()
        self.sigmoid2 = Sigmoid()
        self.sigmoid3 = Sigmoid()
        self.lr = lr
        self.optimizer = SGD([self.weights1, self.weights2, self.weights3], self.lr)

    def forward(self, input):
        self.a1 = self.weights1.forward(input)
        self.b1 = self.sigmoid1.forward(self.a1)
        self.a2 = self.weights2.forward(self.b1)
        self.b2 = self.sigmoid2.forward(self.a2)
        self.a3 = self.weights3.forward(self.b2)
        self.out = self.sigmoid3.forward(self.a3)
        return self.out
```

2.3 Backpropagation

L is loss of forward propagation. $\hat{y}$ is the output of the neural network.

$$\frac{\partial L}{\partial W_3} = \frac{\partial \hat{y}}{\partial W_3}\frac{\partial L}{\partial \hat{y}}$$
$$\frac{\partial L}{\partial W_2} = \frac{\partial b_2}{\partial W_2}\frac{\partial \hat{y}}{\partial b_2}\frac{\partial L}{\partial \hat{y}}$$
$$\frac{\partial L}{\partial W_1} = \frac{\partial b_1}{\partial W_1}\frac{\partial b_2}{\partial b_1}\frac{\partial \hat{y}}{\partial b_2}\frac{\partial L}{\partial \hat{y}}$$

Use Stochastic Gradient Descent(SGD) to update the weights.

$$W_1 = W_1 - \alpha\frac{\partial L}{\partial W_1} \quad W_2 = W_2 - \alpha\frac{\partial L}{\partial W_2} \quad W_3 = W_3 - \alpha\frac{\partial L}{\partial W_3}$$

Every linear layer and activation functions have a backward function to calculate the gradients.

```
def backward(self, gradwrtoutput):
    self.gradW = np.dot(self.input.T, gradwrtoutput)
    return np.dot(gradwrtoutput, self.W.T)
```

We can easily get the gradients of the weights by chain rule.

```
# Backpropagation
dloss_dy = 2*(y-self.Y)
dou_da3 = self.sigmoid3.backward(dloss_dy)
da3_db2 = self.weights3.backward(dou_da3)
db2_da2 = self.sigmoid2.backward(da3_db2)
da2_db1 = self.weights2.backward(db2_da2)
db1_da1 = self.sigmoid1.backward(da2_db1)
da1_dw1 = self.weights1.backward(db1_da1)

# Update Weights
self.optimizer.step()
```

Finally, we can update the weights by SGD.

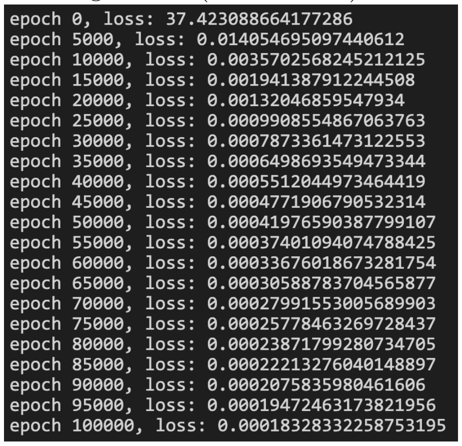
```
class SGD(optimizer):
    def __init__(self, parameters, lr) -> None:
        super().__init__(parameters, lr)

    def step(self):
        for parameter in self.parameters:
            parameter.W -= self.lr * parameter.gradW
```

3 Results of testings

Configurations:
2 Linear Layers
4 Hidden Units
Learning Rate: 0.1
Activation Function: Sigmoid
Optimizer: SGD

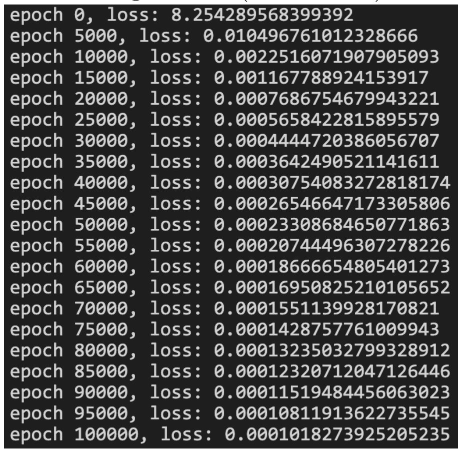
3.1 Screenshot and comparison figure

Training Process(Linear Data):



```
epoch 0, loss: 37.423088664177286
epoch 5000, loss: 0.014054695097440612
epoch 10000, loss: 0.0035702568245212125
epoch 15000, loss: 0.001941387912244508
epoch 20000, loss: 0.0013204685954793 4
epoch 25000, loss: 0.0009908554867063763
epoch 30000, loss: 0.0007873361473122553
epoch 35000, loss: 0.0006498693549473344
epoch 40000, loss: 0.0005512044973464419
epoch 45000, loss: 0.0004771906790532314
epoch 50000, loss: 0.00041976590387799107
epoch 55000, loss: 0.00037401094074788425
epoch 60000, loss: 0.00033676018673281754
epoch 65000, loss: 0.00030588783704565877
epoch 70000, loss: 0.00027991553005689903
epoch 75000, loss: 0.00025778463269728437
epoch 80000, loss: 0.00023871799280734705
epoch 85000, loss: 0.00022213276040148897
epoch 90000, loss: 0.0002075835980461606
epoch 95000, loss: 0.00019472463173821956
epoch 100000, loss: 0.00018328332258753195
```

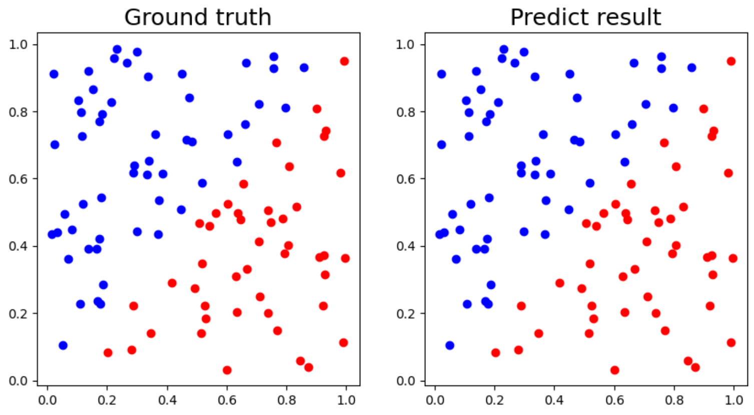
Training Process(XOR Data):

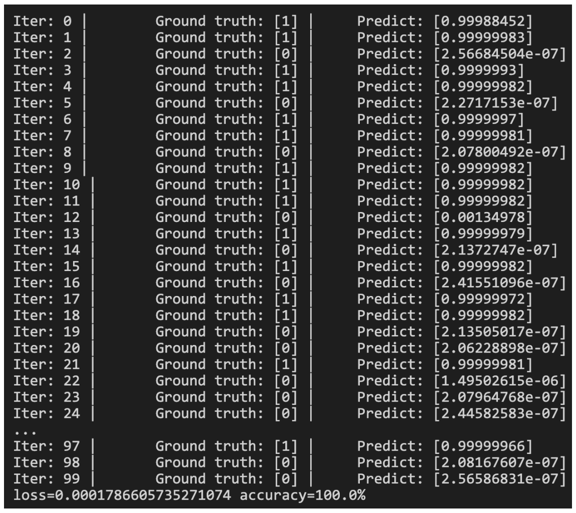


```
epoch 0, loss: 8.254289568399392
epoch 5000, loss: 0.010496761012328666
epoch 10000, loss: 0.0022516071907905093
epoch 15000, loss: 0.001167788924153917
epoch 20000, loss: 0.0007686754679943221
epoch 25000, loss: 0.0005658422815895579
epoch 30000, loss: 0.0004444720386056707
epoch 35000, loss: 0.0003642490521141611
epoch 40000, loss: 0.00030754083272818174
epoch 45000, loss: 0.00026546647173305806
epoch 50000, loss: 0.00023308684650771863
epoch 55000, loss: 0.00020744496307278226
epoch 60000, loss: 0.00018666654805401273
epoch 65000, loss: 0.00016950825210105652
epoch 70000, loss: 0.0001551139928170821
epoch 75000, loss: 0.0001428757761009943
epoch 80000, loss: 0.00013235032799328912
epoch 85000, loss: 0.00012320712047126446
epoch 90000, loss: 0.00011519484456063023
epoch 95000, loss: 0.00010811913622735545
epoch 100000, loss: 0.0001018273925205235
```

3.2 Show the accuracy of your prediction

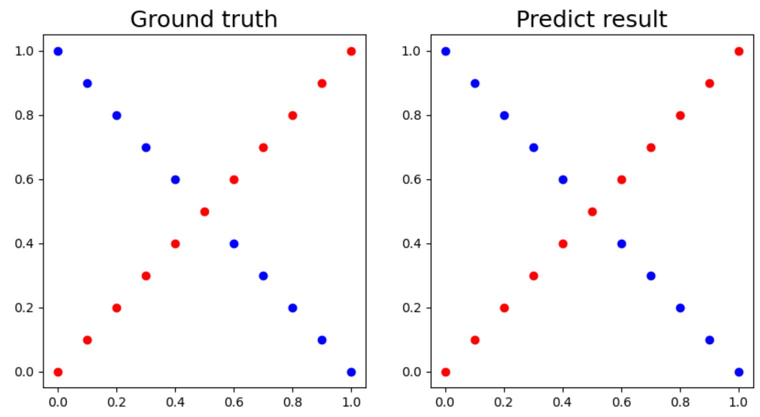
Testing(Linear Data):
Accuracy: 100%
Result and prediction:

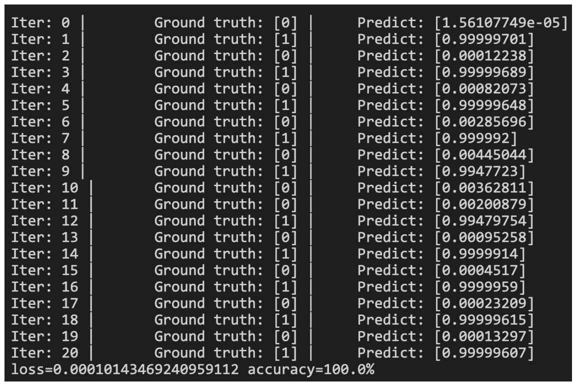




```
Iter: 0 |        Ground truth: [1] |      Predict: [0.99988452]
Iter: 1 |        Ground truth: [1] |      Predict: [0.99999983]
Iter: 2 |        Ground truth: [0] |      Predict: [2.56684504e-07]
Iter: 3 |        Ground truth: [1] |      Predict: [0.9999993]
Iter: 4 |        Ground truth: [1] |      Predict: [0.99999982]
Iter: 5 |        Ground truth: [0] |      Predict: [2.2717153e-07]
Iter: 6 |        Ground truth: [1] |      Predict: [0.9999997]
Iter: 7 |        Ground truth: [1] |      Predict: [0.99999981]
Iter: 8 |        Ground truth: [0] |      Predict: [2.07800492e-07]
Iter: 9 |        Ground truth: [1] |      Predict: [0.99999982]
Iter: 10 |       Ground truth: [1] |      Predict: [0.99999982]
Iter: 11 |       Ground truth: [1] |      Predict: [0.99999982]
Iter: 12 |       Ground truth: [0] |      Predict: [0.00134978]
Iter: 13 |       Ground truth: [1] |      Predict: [0.99999979]
Iter: 14 |       Ground truth: [0] |      Predict: [2.1372747e-07]
Iter: 15 |       Ground truth: [1] |      Predict: [0.99999982]
Iter: 16 |       Ground truth: [0] |      Predict: [2.41551096e-07]
Iter: 17 |       Ground truth: [1] |      Predict: [0.99999972]
Iter: 18 |       Ground truth: [1] |      Predict: [0.99999982]
Iter: 19 |       Ground truth: [0] |      Predict: [2.13505017e-07]
Iter: 20 |       Ground truth: [0] |      Predict: [2.06228898e-07]
Iter: 21 |       Ground truth: [1] |      Predict: [0.99999981]
Iter: 22 |       Ground truth: [0] |      Predict: [1.49502615e-06]
Iter: 23 |       Ground truth: [0] |      Predict: [2.07964768e-07]
Iter: 24 |       Ground truth: [0] |      Predict: [2.44582583e-07]
...
Iter: 97 |       Ground truth: [1] |      Predict: [0.99999966]
Iter: 98 |       Ground truth: [0] |      Predict: [2.08167607e-07]
Iter: 99 |       Ground truth: [0] |      Predict: [2.56586831e-07]
loss=0.0001786605735271074 accuracy=100.0%
```

Testing(XOR Data):
Accuracy: 100%
Result and prediction:

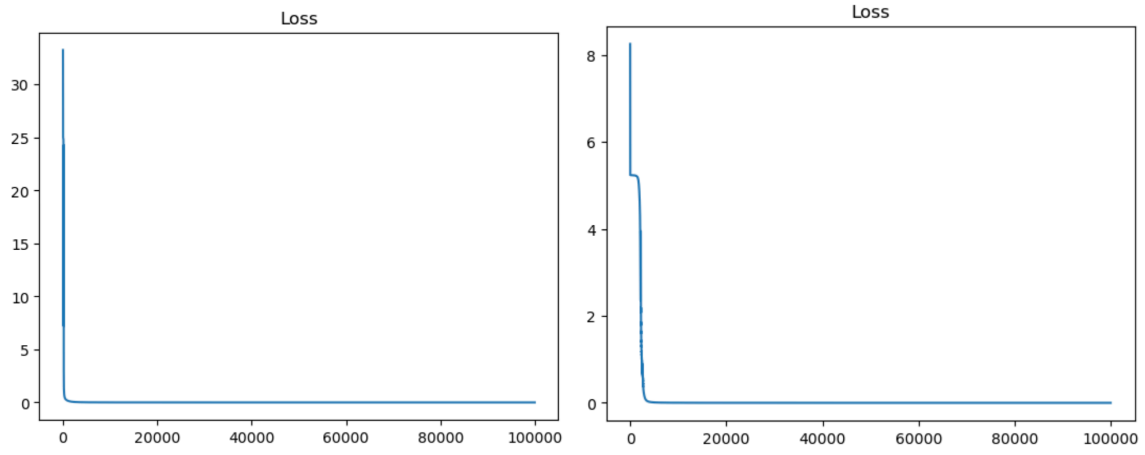




```
Iter: 0 |       Ground truth: [0] |     Predict: [1.56107749e-05]
Iter: 1 |       Ground truth: [1] |     Predict: [0.99999701]
Iter: 2 |       Ground truth: [0] |     Predict: [0.00012238]
Iter: 3 |       Ground truth: [1] |     Predict: [0.99999689]
Iter: 4 |       Ground truth: [0] |     Predict: [0.00082073]
Iter: 5 |       Ground truth: [1] |     Predict: [0.99999648]
Iter: 6 |       Ground truth: [0] |     Predict: [0.00285696]
Iter: 7 |       Ground truth: [1] |     Predict: [0.999992]
Iter: 8 |       Ground truth: [0] |     Predict: [0.00445044]
Iter: 9 |       Ground truth: [1] |     Predict: [0.9947723]
Iter: 10 |      Ground truth: [0] |     Predict: [0.00362811]
Iter: 11 |      Ground truth: [0] |     Predict: [0.00200879]
Iter: 12 |      Ground truth: [1] |     Predict: [0.99479754]
Iter: 13 |      Ground truth: [0] |     Predict: [0.00095258]
Iter: 14 |      Ground truth: [1] |     Predict: [0.9999914]
Iter: 15 |      Ground truth: [0] |     Predict: [0.0004517]
Iter: 16 |      Ground truth: [1] |     Predict: [0.9999959]
Iter: 17 |      Ground truth: [0] |     Predict: [0.00023209]
Iter: 18 |      Ground truth: [1] |     Predict: [0.99999615]
Iter: 19 |      Ground truth: [0] |     Predict: [0.00013297]
Iter: 20 |      Ground truth: [1] |     Predict: [0.99999607]
loss=0.00010143469240959112 accuracy=100.0%
```

3.3 Learning curve (loss, epoch curve)

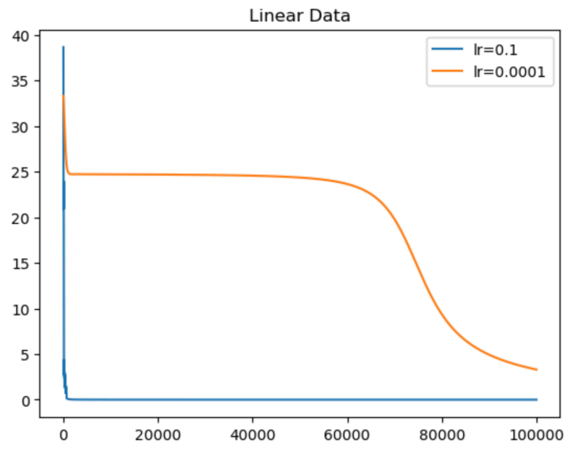
Learning curve (Left: Linear Data / Right: XOR data)



4 Discussion

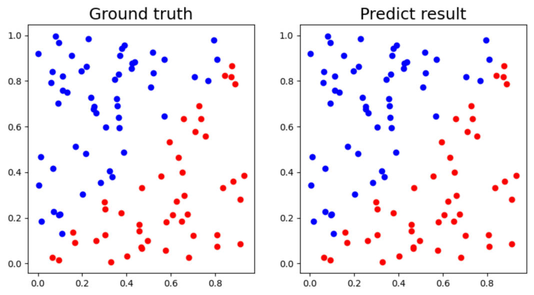
4.1 Different learning rates

Linear Data:

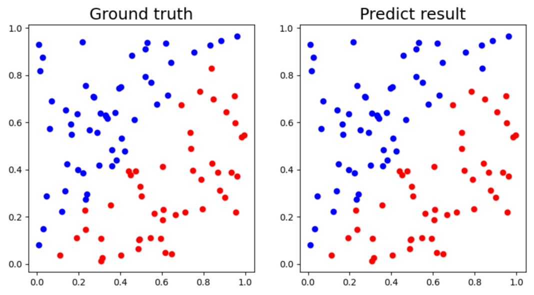


We can find that the learning rate is too small, the loss will decrease slowly. The accuracy of lr=0.1 is 100% and the accuracy of lr=0.0001 is 99%.

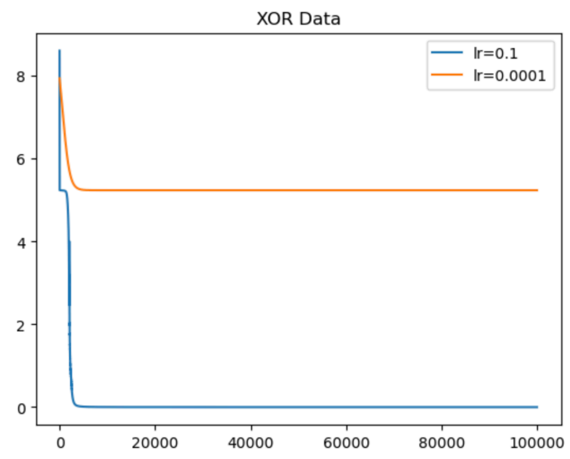
Prediction result(lr=0.1):



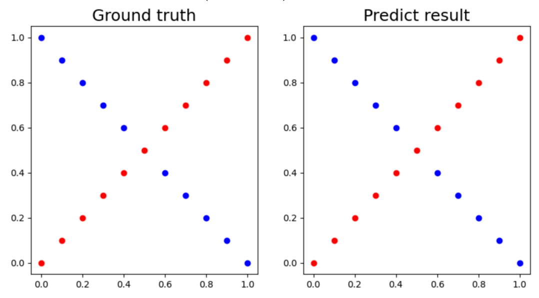
Prediction result(lr=0.0001):



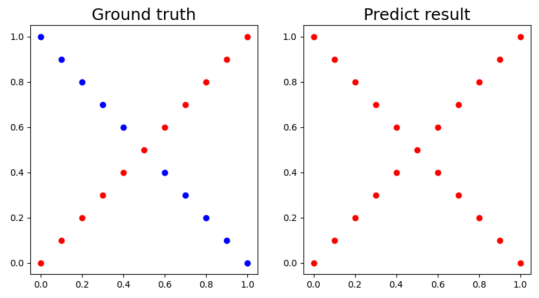
XOR Data:



We can find that the learning rate is too small, the loss will decrease slowly. The accuracy of lr=0.1 is 100% and the accuracy of lr=0.0001 is 52%.
Prediction result(lr=0.1):

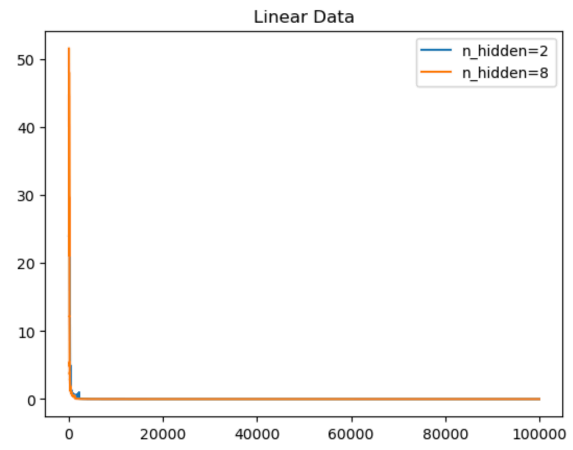


Prediction result(lr=0.0001):



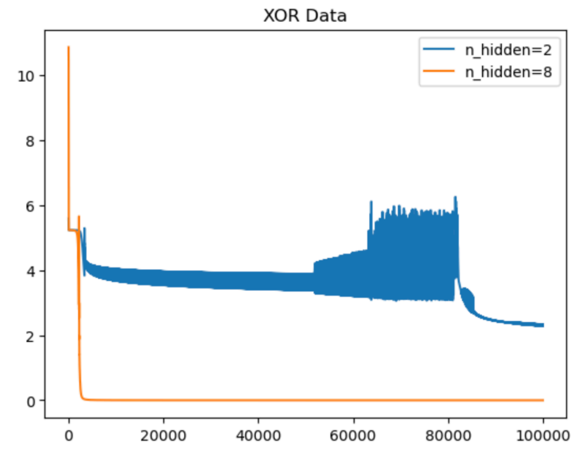
4.2 Different numbers of hidden units

We compare the loss and accuracy between 2 hidden units and 8 hidden units.
Linear Data:



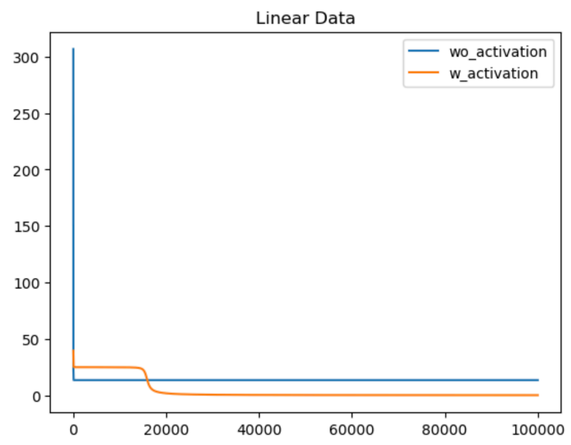
Both accuracies are 100%. Numbers of hidden units do not affect the accuracy a lot.

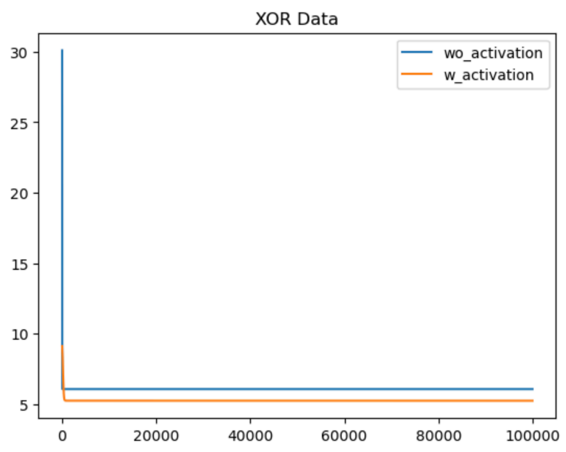
XOR Data:



When number of hidden units is 2, the loss can not converge very well and the accuracy is only 72%. But when the number of hidden units increase to 8, the loss can converge well and the accuracy is 100%.

4.3 Try without activation functions



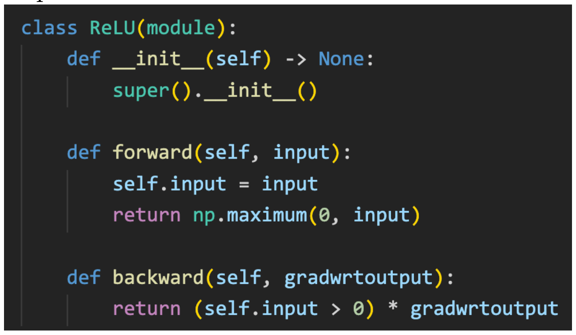


The model can not learn very well without activation functions. I tuned the learning rate to 0.01 since if the learning rate is too large, the loss will explode without activation function. The accuracy of model without activation functions is only 88% under linear data and 33% under XOR data.

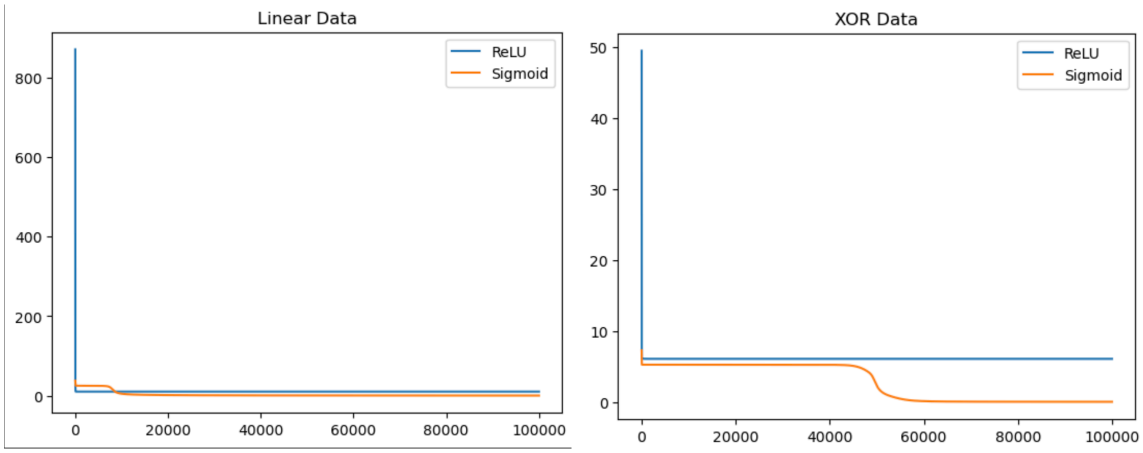
5 Extra

5.1 Different activation functions

Implement ReLU activation function.



```python
class ReLU(module):
    def __init__(self) -> None:
        super().__init__()

    def forward(self, input):
        self.input = input
        return np.maximum(0, input)

    def backward(self, gradwrtoutput):
        return (self.input > 0) * gradwrtoutput
```



The loss of ReLU activation function is higher than Sigmoid activation function. The accuracy of ReLU activation function is 86% under linear data and 33% under XOR data. I suggest that ReLU can not bound the output to [0,1], so the model is harder to learn.

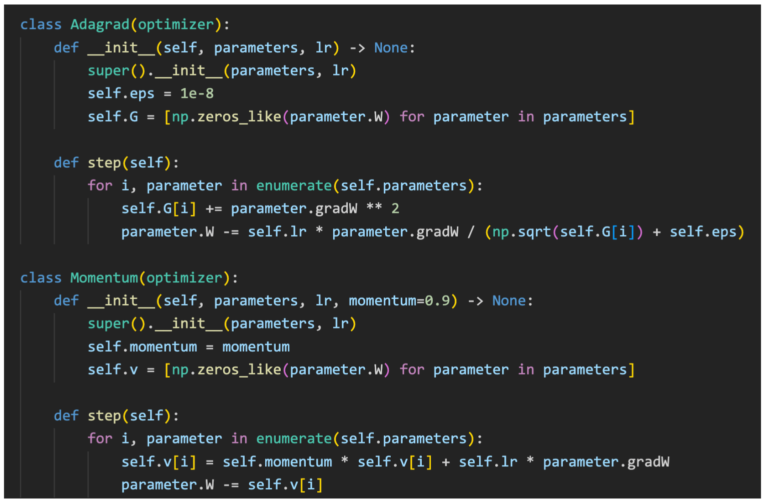
5.2 Different optimizers

Implement Adagrad and Momentum optimizers.
Momentum optimizer:

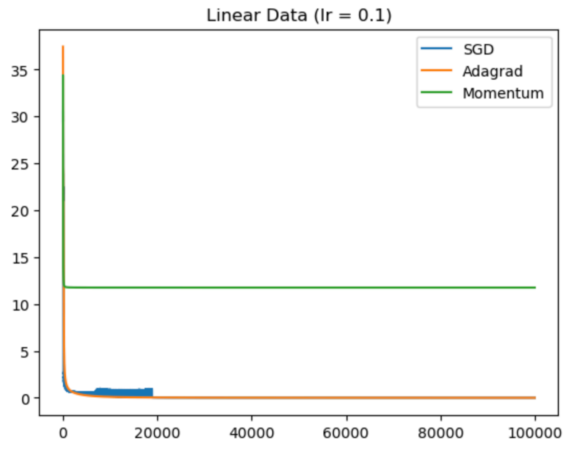
$$v_{t+1} = \beta v_t + \nabla L \quad W_{t+1} = W_t - \alpha v_{t+1}$$

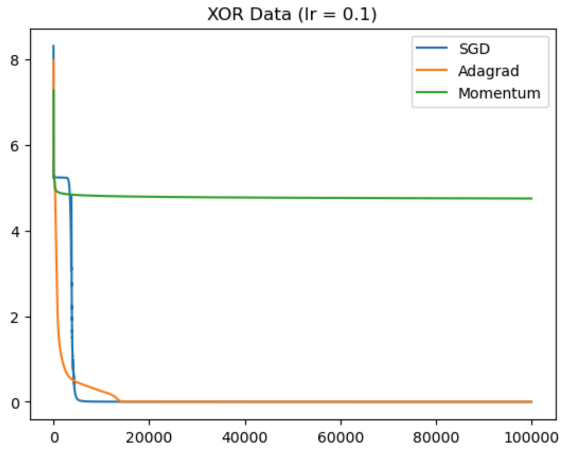
Adagrad optimizer:

$$G_{t+1} = G_t + (\nabla L)^2 \quad W_{t+1} = W_t - \frac{\alpha}{\sqrt{G_{t+1} + \epsilon}} \nabla L$$

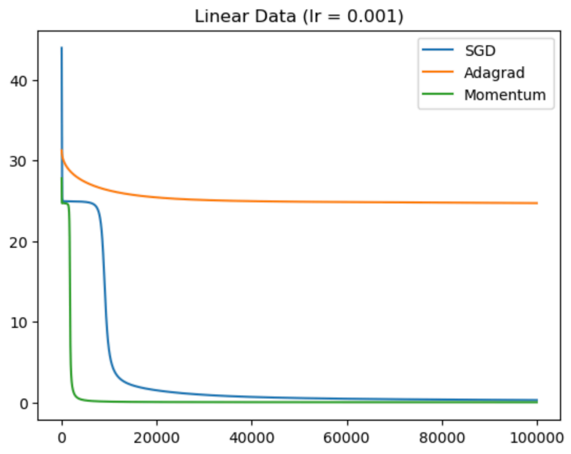

```python
class Adagrad(optimizer):
    def __init__(self, parameters, lr) -> None:
        super().__init__(parameters, lr)
        self.eps = 1e-8
        self.G = [np.zeros_like(parameter.W) for parameter in parameters]

    def step(self):
        for i, parameter in enumerate(self.parameters):
            self.G[i] += parameter.gradW ** 2
            parameter.W -= self.lr * parameter.gradW / (np.sqrt(self.G[i]) + self.eps)

class Momentum(optimizer):
    def __init__(self, parameters, lr, momentum=0.9) -> None:
        super().__init__(parameters, lr)
        self.momentum = momentum
        self.v = [np.zeros_like(parameter.W) for parameter in parameters]

    def step(self):
        for i, parameter in enumerate(self.parameters):
            self.v[i] = self.momentum * self.v[i] + self.lr * parameter.gradW
            parameter.W -= self.v[i]
```

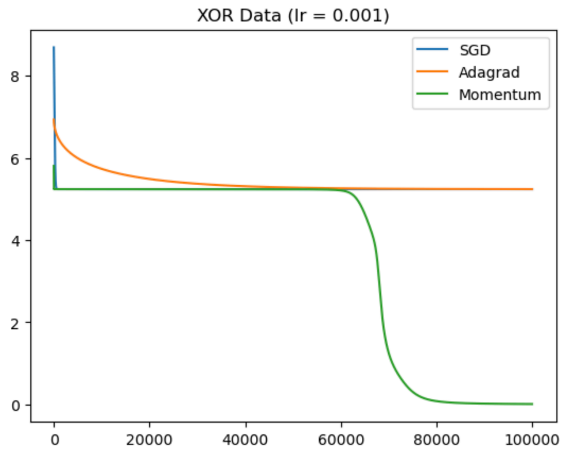
When the learning rate is 0.1, SGD and Adagrad can converge quikly and the accuracy is 100% or near 100%. But Momentum optimizer can not converge well and the accuracies are only 55% and 52%.





When the learning rate is 0.001, Momentum can converge well under two datasets and the accuracy is 100% or near 100%. While SGD can perform well under linear data but not well under XOR data. Adagrad can not converge well under two datasets. The accuracy of SGD is 100% under linear data and 52% under xor data. The accuracy of Adagrad is 50% and 52%.


Linear Data (lr = 0.001)
SGD
Adagrad
Momentum


XOR Data (lr = 0.001)
SGD
Adagrad
Momentum