DLP Lab1 Report

> 310551178 資科工碩 穆冠蓁

1. Introduction

This lab is to implement a neural network without using PyTorch or Tensorflow, we predict the answer and compute the loss with the ground truth and backpropogate it to update the weight, do it iteratively to predict the answer more accurately.Beside, we do experiment on different learning rate, hidden units, optimizers, activation functions to observe the difference.

2. Experiment setups

A. Sigmoid functions

Above is the sigmoid function and derivative of sigmoid function, which was written in `NeuronLayer`.`sigmoid` function is for the forward pass, and `derivativeSigmoid` function is for the backward pass.

```
# Sigmoid function
@staticmethod
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
# derivative sigmoid
@staticmethod
def derivativeSigmoid(y):
    return np.multiply(y, 1.0 - y)
```

B. Neural network

- Architecture of neural network:

```
class NeuralNetwork:
    def __init__(self,
                 hiddenLayerNum = 2,
                 neuronNum = 4,
                 lr = 0.001,
                 epoch = 10000,
                 activationType = 'None',
                 optimizerType = 'sgd'):

    def forward(self, inputs):

    def backward(self, loss):

    def update(self): # Optimizer

    # Loss function
    def MSE(self, yHat, y):

    def MSE_derivative(self, prediction, groundtruth):

    # train
```

```
    def train(self, x, y):

    # Prediction
    def prediction(self, x):

    # Compute accuracy
    def accuracy(self, groundTruth, predict):

    # Data visualize
    def show_result(self, x, y):
```

`__init__` :initialize some parameters, and also initialize the input layer, hidden layer and output layer.

`forward` : forward function

`backward` : backward function

`update`: update the weight

`MSE` , `MSE_derivatve` : the loss function and the derivative of loss function

`train` : train the define epochs

`prediction` : predict the testing data

`accuracy` : Compute the accuracy, loss

`show_result` : visualize the predict result and the ground truth, and also the learning curve

- Architecture of layer:

```
class NeuronLayer:
    def __init__(self,
                 in_channel,
                 out_channel,
                 lr,
                 activation = 'sigmoid',
                 optimizer = 'sgd'):

    def forward(self,inputs):

    def backward(self, derivative):

    def update(self):
```

`__init__` : initialize the input channel# ,output channel#, learning rate, activation function type and optimizer type

`forward` : compute forward gradient and choose different activation function, such as sigmoid, tanh, ReLU, leaky ReLU.

`backward` : compute the backward gradient.

`update`: multiply forward gradient and backward gradient, choose different optimizer to update the weight.

C. Backpropagation

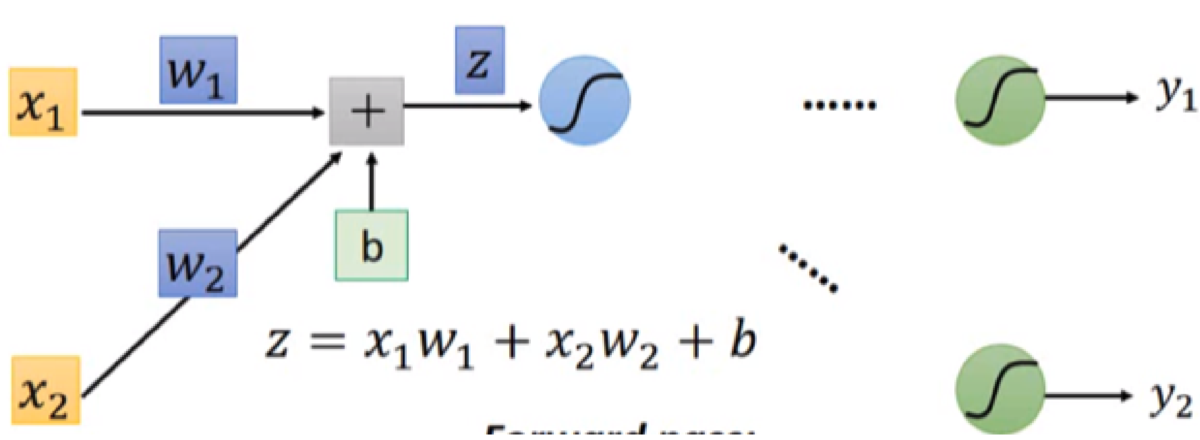
```
def forward(self,inputs):
    self.forwardGrad = np.append(inputs, np.ones((inputs.shape[0], 1)), axis=1)
    if self.activateFunc == 'sigmoid':
        self.output = self.sigmoid(np.matmul(self.forwardGrad, self.weight))
    elif self.activateFunc == 'tanh':
        self.output = self.tanh(np.matmul(self.forwardGrad, self.weight))
    elif self.activateFunc == 'relu':
        self.output = self.ReLU(np.matmul(self.forwardGrad, self.weight))
    elif self.activateFunc == 'lrelu':
        self.output = self.LReLU(np.matmul(self.forwardGrad, self.weight))
    else: # Without activation function
        self.output = np.matmul(self.forwardGrad, self.weight)

    return self.output
```

```
def backward(self, derivative):
    if self.activateFunc == 'sigmoid':
        self.backwardGrad = np.multiply(self.derivativeSigmoid(self.output), derivative)
    elif self.activateFunc == 'tanh':
        self.backwardGrad = np.multiply(self.derivativeTanh(self.output), derivative)
    elif self.activateFunc == 'relu':
        self.backwardGrad = np.multiply(self.derivativeReLU(self.output), derivative)
    elif self.activateFunc == 'lrelu':
        self.backwardGrad = np.multiply(self.derivativeLReLU(self.output), derivative)
    else:# Without activation function
        self.backwardGrad = derivative
    return np.matmul(self.backwardGrad, self.weight[:-1].T)
```

```
def update(self):
    grad = np.matmul(self.forwardGrad.T, self.backwardGrad)
    if self.optimizerFunc == 'sgd':
        deltaWeight = -self.lr * grad
    elif self.optimizerFunc == 'momentum':
        self.momentum = 0.9 * self.momentum - self.lr * grad
        deltaWeight = self.momentum
    elif self.optimizerFunc == 'Adagrad':
        self.sum_of_squares_of_gradients += np.square(grad)
        deltaWeight = -self.lr * grad / np.sqrt(self.sum_of_squares_of_gradients + 1e-8)
    self.weight += deltaWeight
    return self.weight
```

We have seperate the backpropagation into three phase, which include forwardpass, backwardpass at last update the weight with the update function.

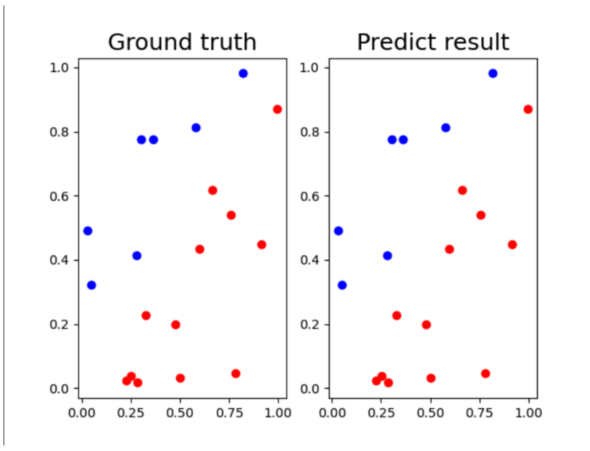
The loss function is define as $L(\theta) = \sum C^n(\theta)$, to do partial derivative on the loss function $\frac{\partial L(\theta)}{\partial w} = \sum \frac{\partial C^n(\theta)}{\partial w}$, $\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial C}{\partial z}$(chain rule).We can know that $\frac{\partial z}{\partial w}$ is the forward pass, and $\frac{\partial C}{\partial z}$ is the backward pass.

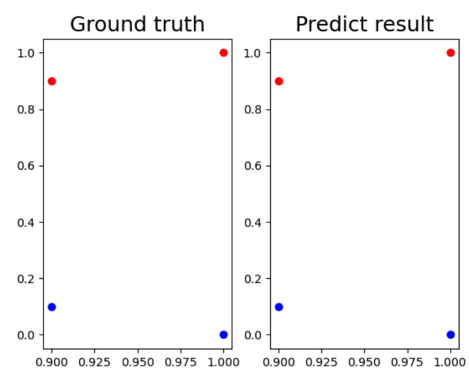


3. Result of your testing

A. Screenshot and comparison figure

- Linear
- XOR





B. Show the accuracy of your prediction

- Linear

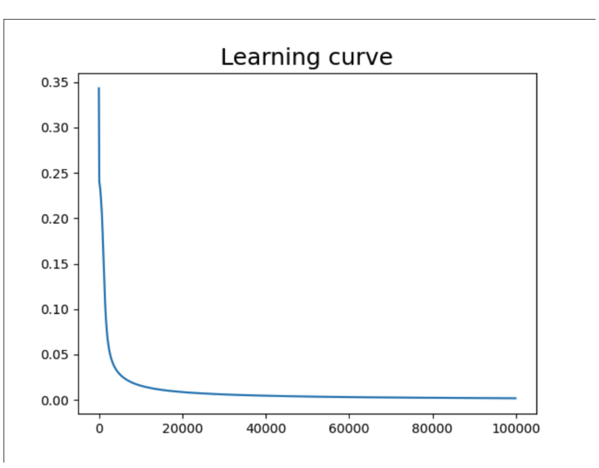
```
Epoch = 99400, loss = 0.0018520875017863313
Epoch = 99500, loss = 0.0018499817570448134
Epoch = 99600, loss = 0.0018478797238284917
Epoch = 99700, loss = 0.0018457813915926844
Epoch = 99800, loss = 0.0018436867498360376
Epoch = 99900, loss = 0.0018415957881004947
---------- Testing ----------
Iter = 1   Ground truth = [1] prediction = [0.99999258]
Iter = 2   Ground truth = [0] prediction = [1.93701341e-05]
Iter = 3   Ground truth = [0] prediction = [4.07416333e-05]
Iter = 4   Ground truth = [1] prediction = [0.9999842]
Iter = 5   Ground truth = [1] prediction = [0.99999205]
Iter = 6   Ground truth = [0] prediction = [1.29907028e-05]
Iter = 7   Ground truth = [0] prediction = [7.09483929e-05]
Iter = 8   Ground truth = [0] prediction = [1.78987373e-05]
Iter = 9   Ground truth = [1] prediction = [0.99999279]
Iter = 10   Ground truth = [1] prediction = [0.99994541]
Iter = 11   Ground truth = [0] prediction = [2.93773631e-05]
Iter = 12   Ground truth = [0] prediction = [2.67059868e-05]
Iter = 13   Ground truth = [1] prediction = [0.99998927]
Iter = 14   Ground truth = [0] prediction = [1.21839562e-05]
Iter = 15   Ground truth = [0] prediction = [2.27967025e-05]
Iter = 16   Ground truth = [1] prediction = [0.99992425]
Iter = 17   Ground truth = [0] prediction = [0.00310542]
Iter = 18   Ground truth = [0] prediction = [0.0002344]
Iter = 19   Ground truth = [0] prediction = [1.34572409e-05]
loss = 9.717796126129775e-06, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

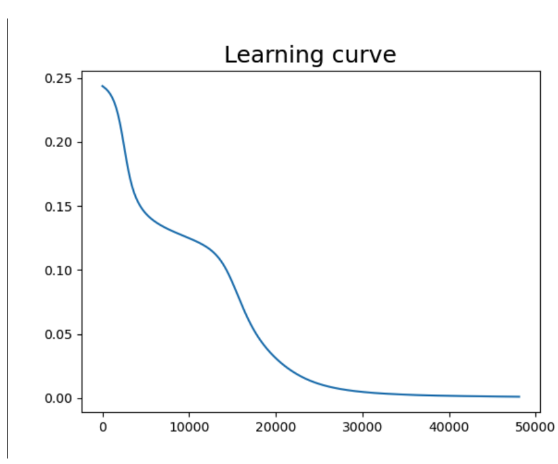
- XOR

```
Epoch = 46500, loss = 0.0010951821219314453
Epoch = 46600, loss = 0.001088958974687036
Epoch = 46700, loss = 0.0010827959705147138
Epoch = 46800, loss = 0.0010766923087859874
Epoch = 46900, loss = 0.001070647202267194
Epoch = 47000, loss = 0.0010646598768522392
Epoch = 47100, loss = 0.0010587295713014124
Epoch = 47200, loss = 0.0010528555369862412
Epoch = 47300, loss = 0.0010470370376401419
Epoch = 47400, loss = 0.0010412733491147776
Epoch = 47500, loss = 0.001035563759141947
Epoch = 47600, loss = 0.001029907567100915
Epoch = 47700, loss = 0.0010243040837908846
Epoch = 47800, loss = 0.0010187526312087085
Epoch = 47900, loss = 0.0010132525423315953
Epoch = 48000, loss = 0.0010078031609045772
Epoch = 48100, loss = 0.0010024038412328362
---------- Testing ----------
Iter = 1   Ground truth = [0] prediction = [0.00391673]
Iter = 2   Ground truth = [1] prediction = [0.9997894]
Iter = 3   Ground truth = [0] prediction = [0.00245673]
Iter = 4   Ground truth = [1] prediction = [0.99980854]
loss = 2.1457295012773997e-05, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

C. Learning curve(loss, epoch curve)

- Linear
- XOR





D. Discussion of the comparison

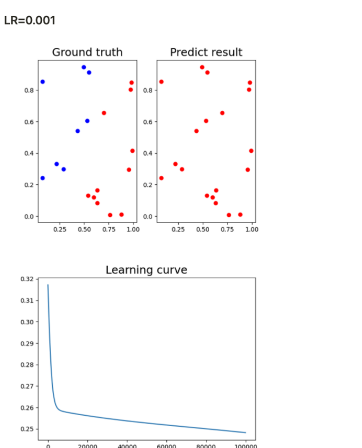
We can see the comparison between the ground truth and the predict result, accuracy, learning curve.We can get 100% of accuracy in both Linear and XOR data type, and the mainly difference is the learning curve, which XOR data converge in the early training step and the training loss is smaller than 0.01 that we stop training.

4. Discussion

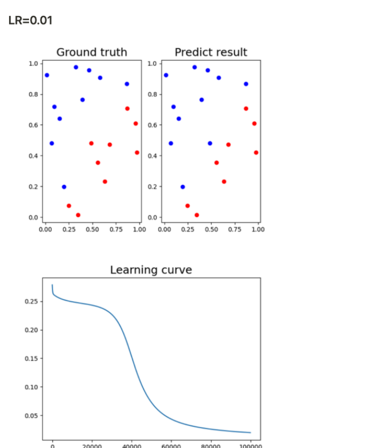
A. Try different learning rates

- Linear

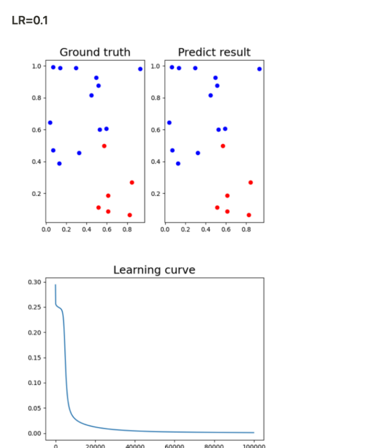
  We can see while we set the learning rate to 0.1 has the best accuracy in the experiment.




```
# Result
loss = 4.692713046243734, accuracy = 57.89473684210527%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```



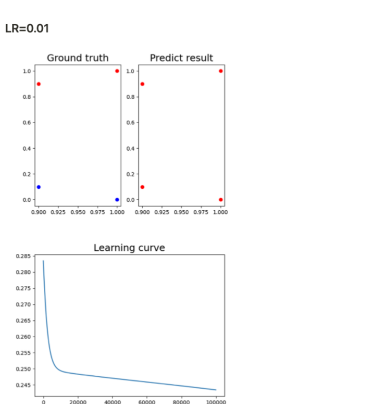

```
# Result
loss = 0.6974396455509467, accuracy = 94.73684210526315%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```



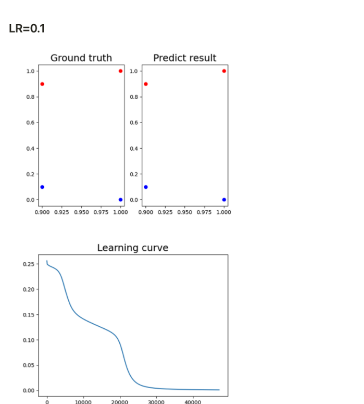

```
# Result
loss = 0.009487308536608272, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

- XOR

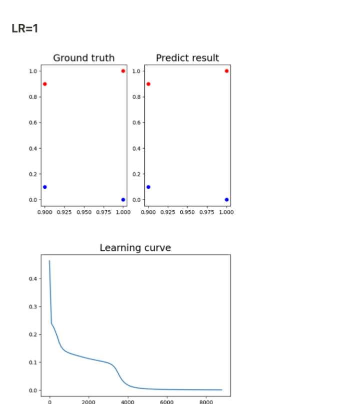
  We found that while the learning rate is too small, we will get the worse accuracy during training, so we have set the learning rate to a bigger number.




```
# Result
loss = 1.8131405107566714, accuracy = 50.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```




```
# Result
loss = 0.004673090638991842, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```



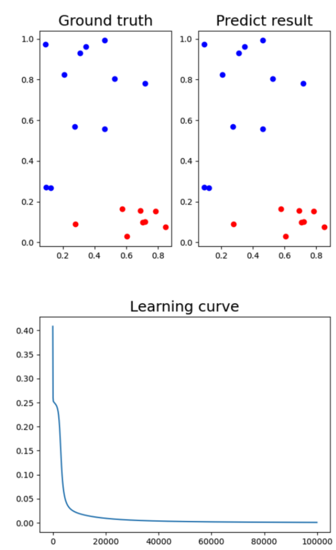

```
# Result
loss = 0.0030521511160174833, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

B. Try different numbers of hidden units

- Linear

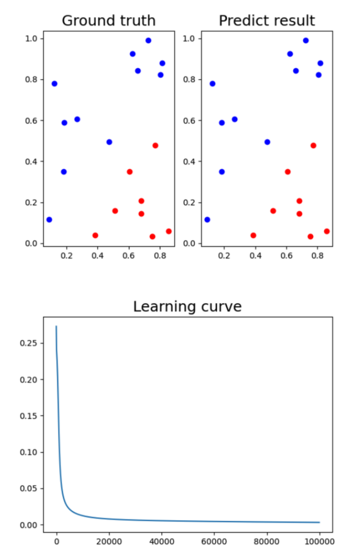
  We get the similar result while we try on bigger hidden units.

Hidden units = 4



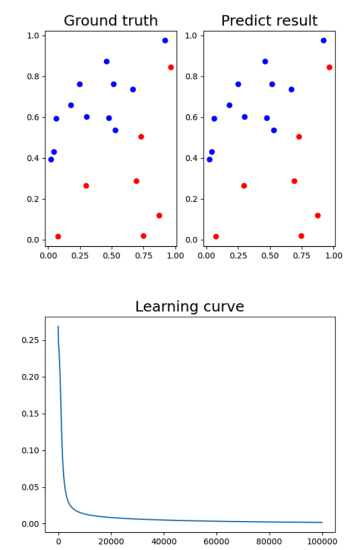

```
loss = 1.4393798291043496e-05, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

Hidden units = 8




```
loss = 0.007952638353260497, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

Hidden units = 12

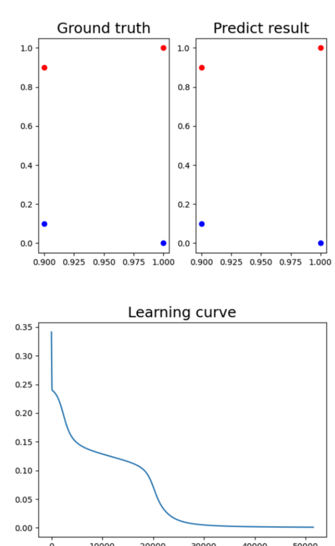



```
loss = 0.04198288588589966, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

- XOR

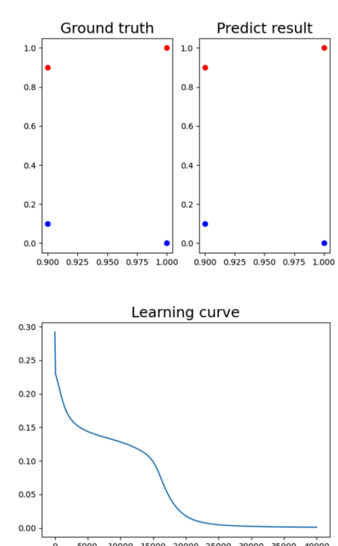
  We have see a special phenomena in the XOR data, we can find that when the hidden units increase it converge earlier.

Hidden units = 4



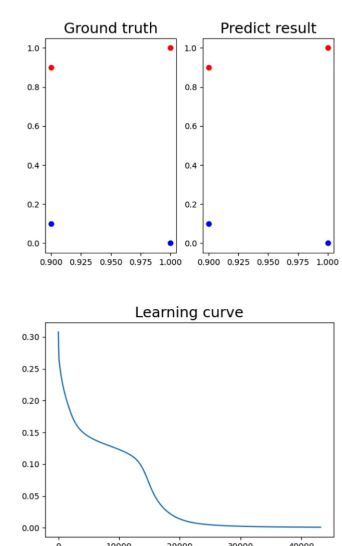

```
loss = 0.0014680883585813256, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

Hidden units = 8




```
loss = 0.0008349199857406491, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

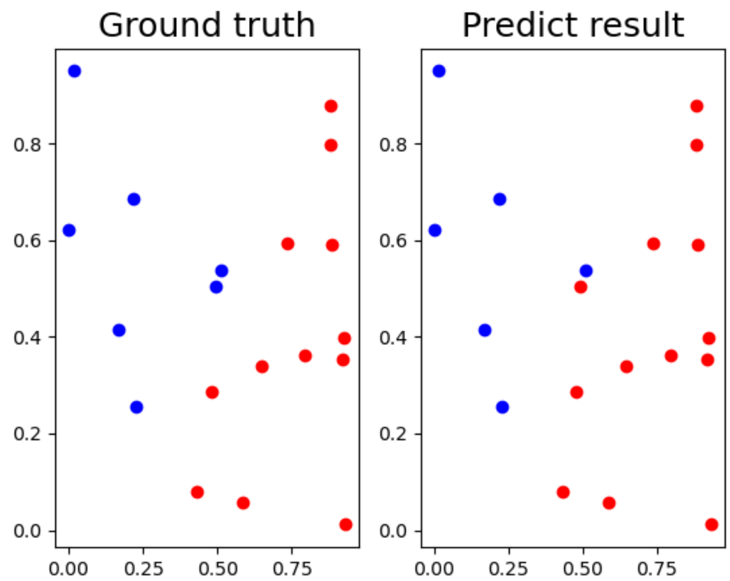
Hidden units = 12

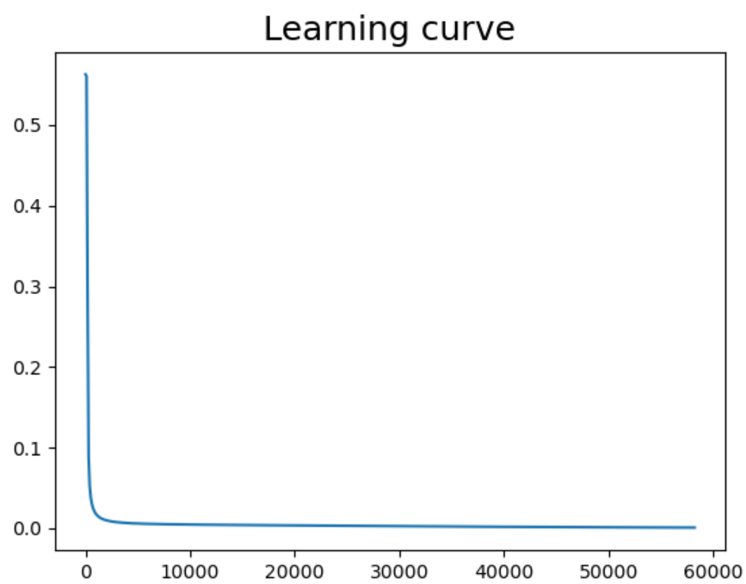



```
loss = 0.0010202650648179665, accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

C. Try without activation functions

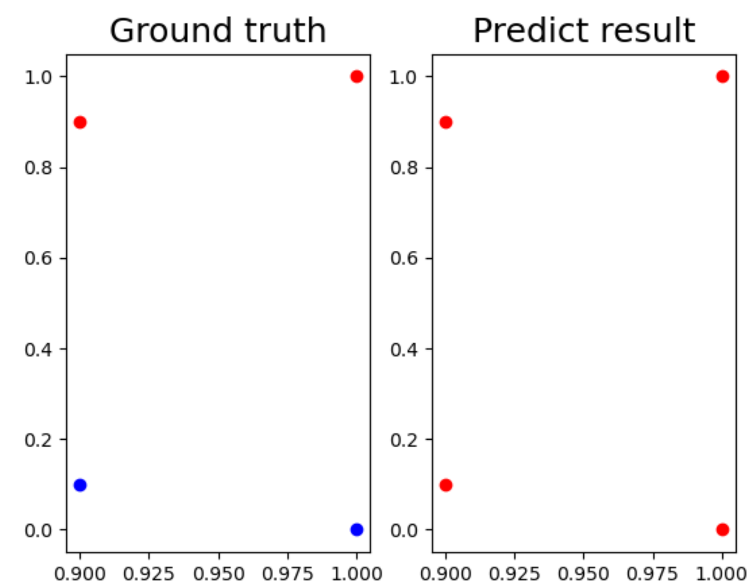
- Linear

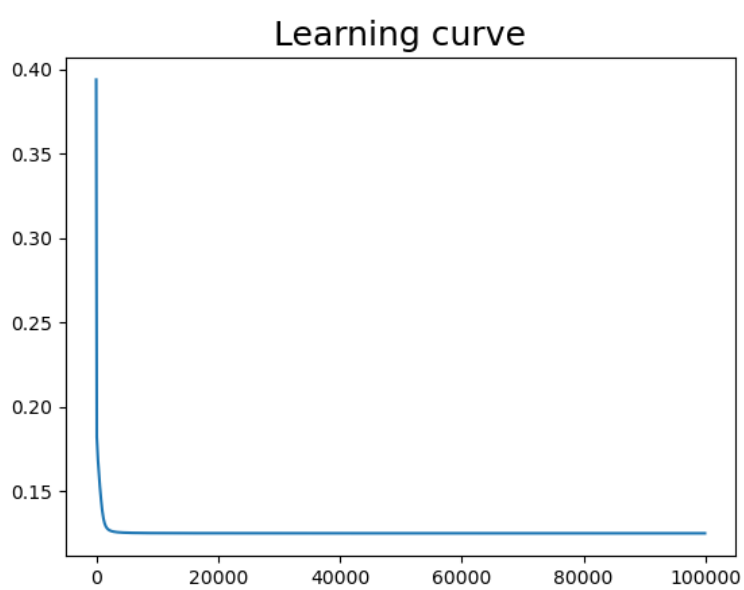





```
loss = 0.41333500330564377, accuracy = 94.73684210526315%
Data type =  Linear
Activation function = none
Optimizer type =  sgd
```

- XOR






```
loss = 2.000022951323706, accuracy = 50.0%
Data type =  XOR
Activation function = none
Optimizer type =  sgd
```

D. anything you want to present

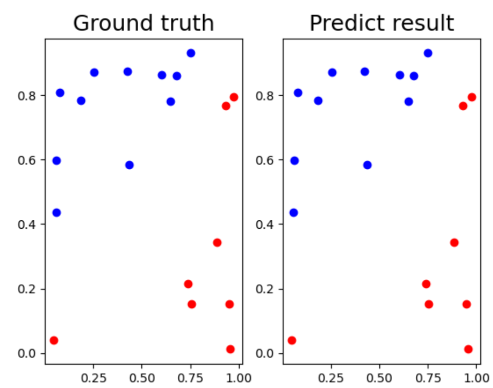
During the experiment, I found out that without activation function, the loss we become extremely big(NaN).To tackle this problem, I will put an activation function in the output layer, that will make my accuracy of linear data become 94% , but the XOR data still has a low accuracy 50%.

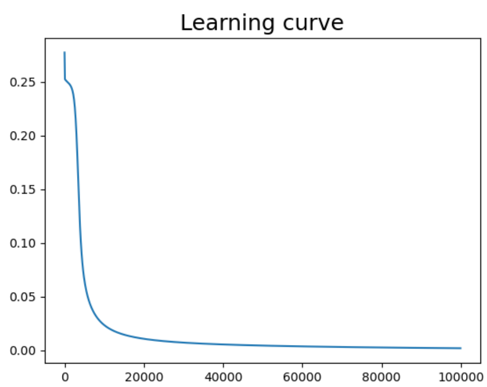
5. Extra

A. Implement different optimizers

- Linear

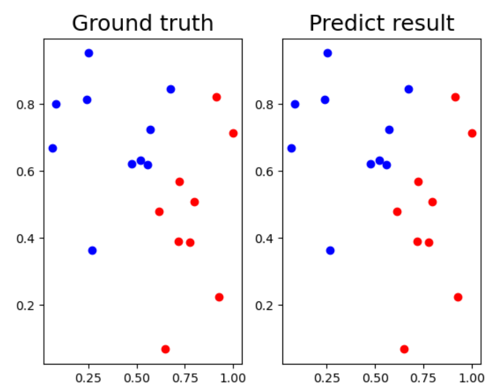
SGD

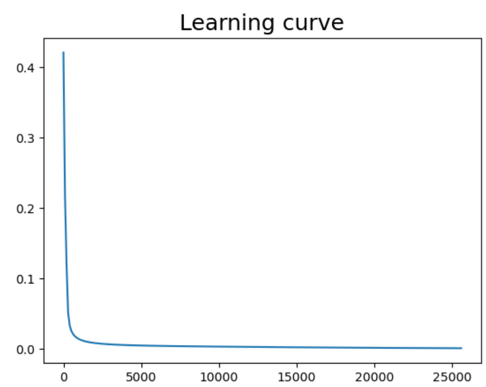





```
loss = 0.1893440264923813,
accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```

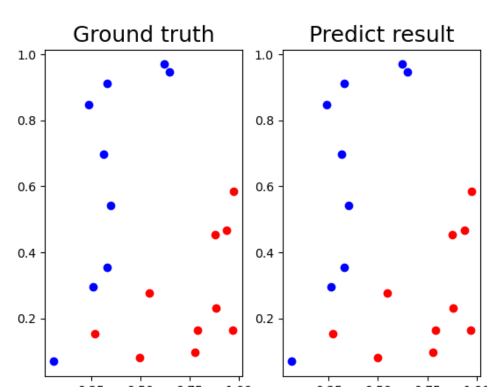
momentum

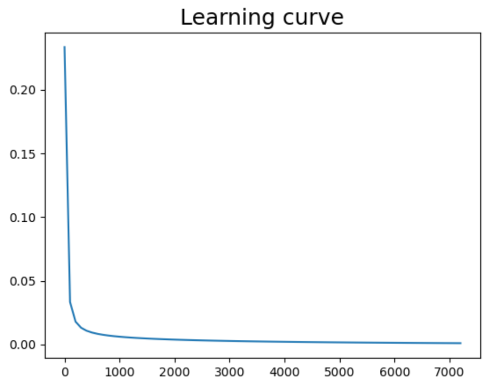





```
loss = 1.2727034051908915e-07,
accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  momentum
```

Adagrad

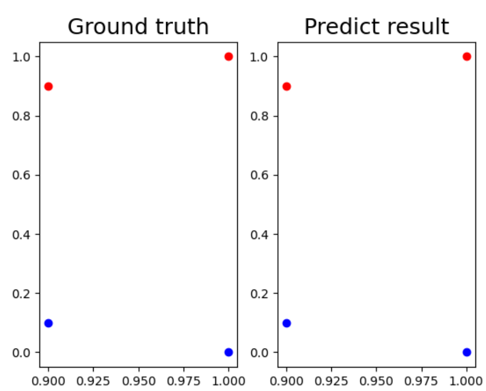


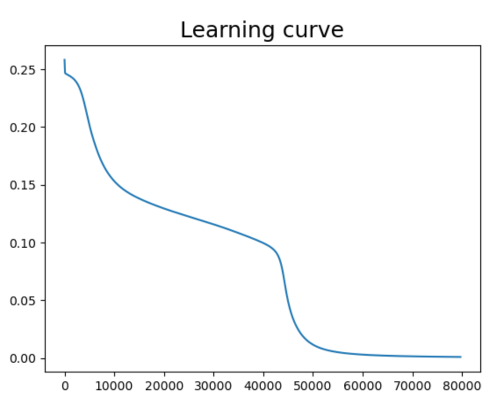



```
loss = 0.001120287293556949, accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  Adagrad
```

- XOR

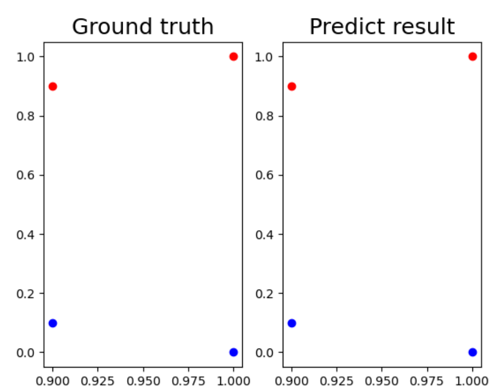
SGD

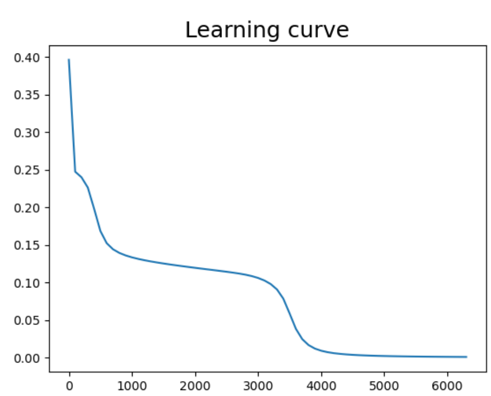





```
loss = 0.001034427808599671,
accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

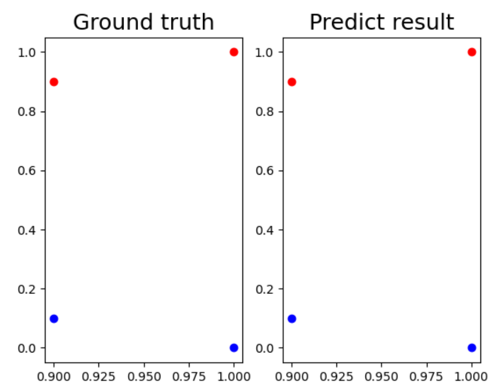
momentum

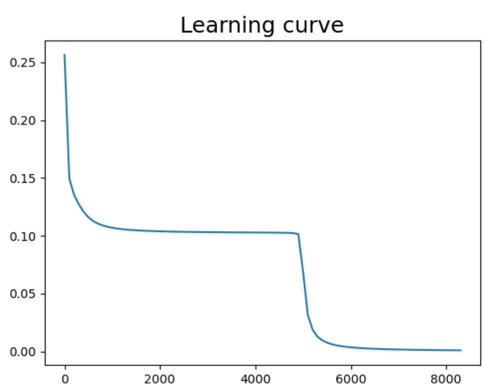





```
loss = 0.002720251058816887,
accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  momentum
```

Adagrad





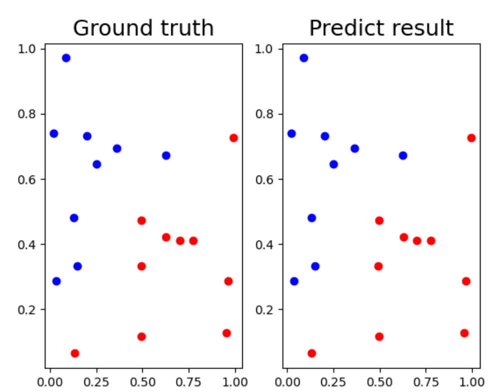

```
loss = 0.006956403461242546,
accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  Adagrad
```

I found that the the converge speed: SGD<momentum<Adagrad, we can also observe that both momentum and adagrad has a smoother learning curve than SGD, because momentum and adagrad have special technique to prevent oscilliation and enhance converge speed.

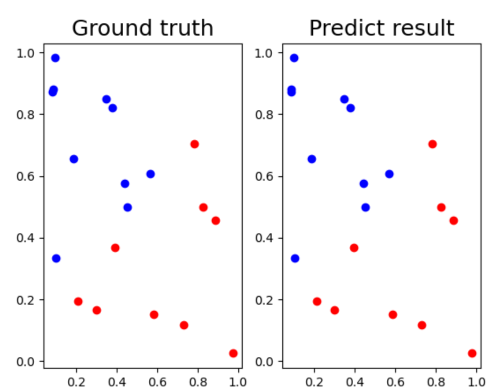
B. Implement different activation functions

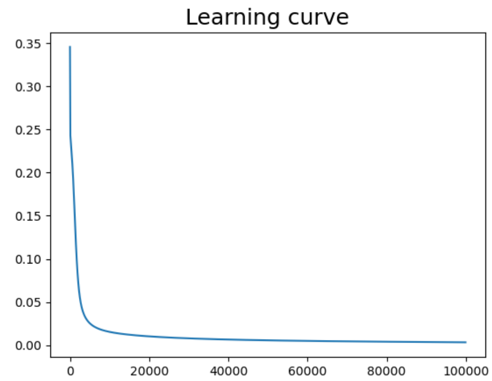
- Linear

Sigmoid

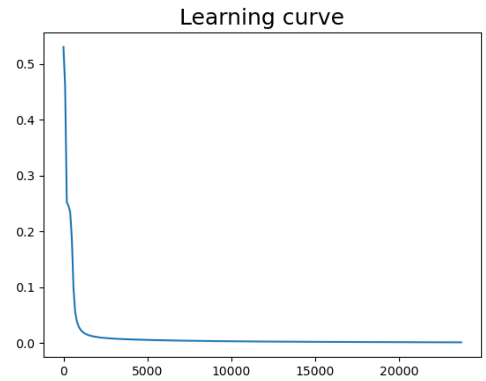


tanh



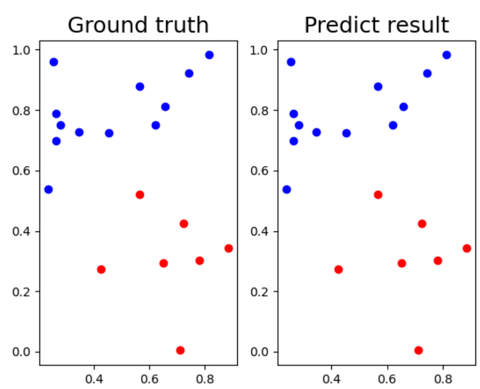


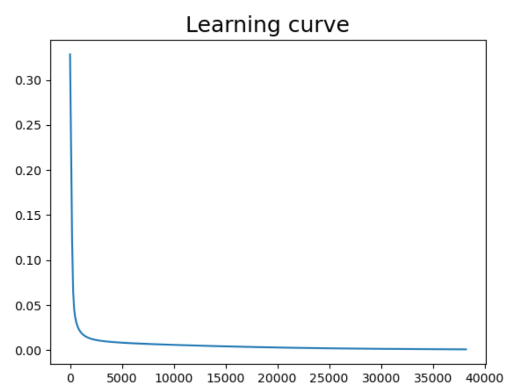

```
loss = 0.054084329293294924,
accuracy = 100.0%
Data type =  Linear
Activation function = sigmoid
Optimizer type =  sgd
```




```
loss = 0.013832146409289 24,
accuracy = 100.0%
Data type =  Linear
Activation function = tanh
Optimizer type =  sgd
```

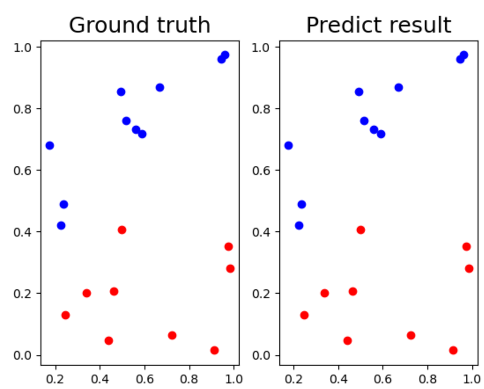
ReLU

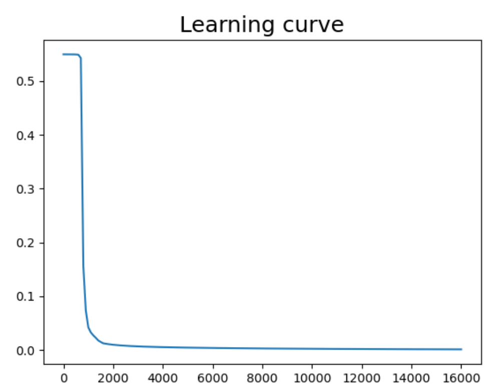





```
loss = 3.583640758924742e-08,
accuracy = 100.0%
Data type =  Linear
Activation function = relu
Optimizer type =  sgd
```

Leaky ReLU



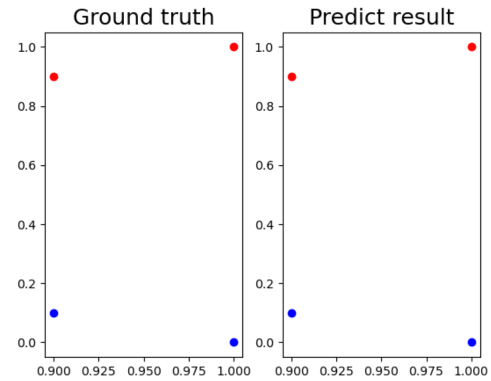


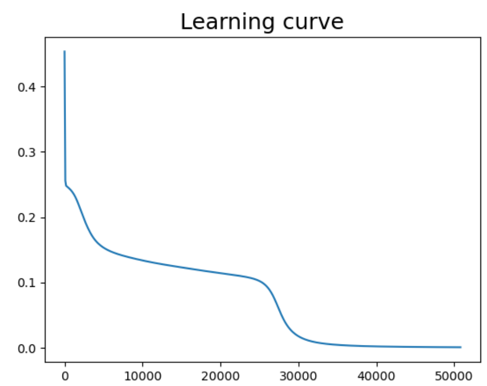

```
loss = 0.015924671984322368,
accuracy = 100.0%
Data type =  Linear
Activation function = lrelu
Optimizer type =  sgd
```

- XOR

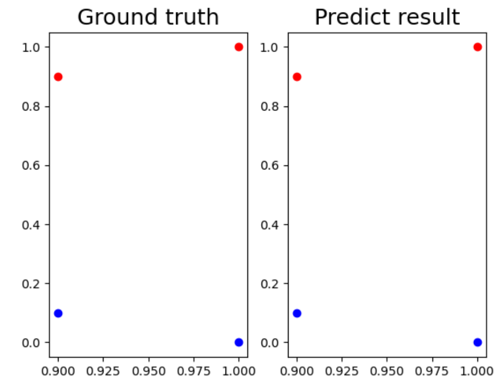
Sigmoid

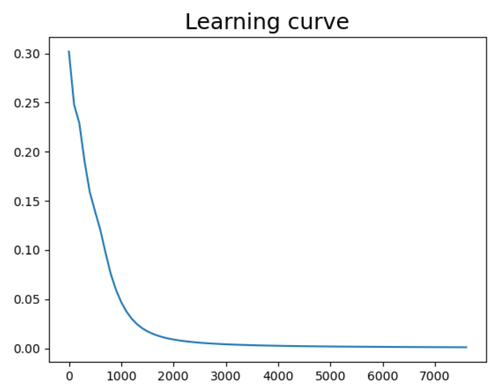
tanh





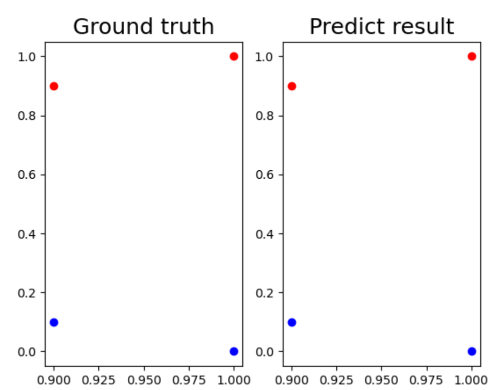

```
loss = 0.0038135679484962667,
accuracy = 100.0%
Data type =  XOR
Activation function = sigmoid
Optimizer type =  sgd
```

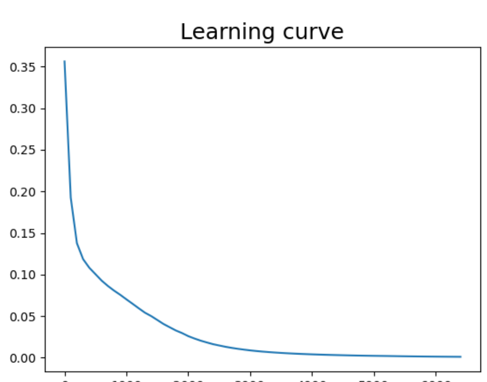




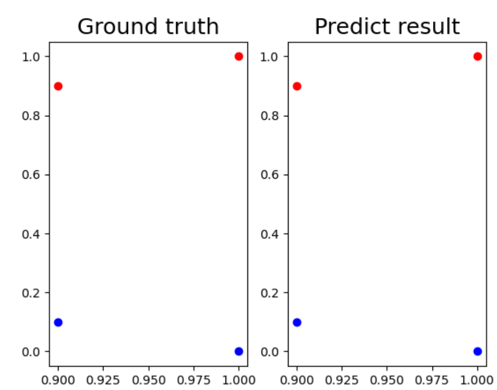

```
loss = 0.003515666064403951,
accuracy = 100.0%
Data type =  XOR
Activation function = tanh
Optimizer type =  sgd
```

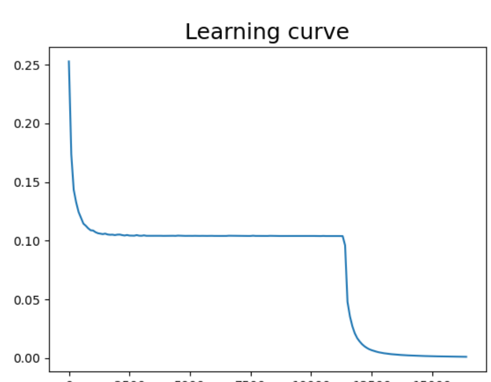
ReLU





Leaky ReLU





```
loss = 1.1422606913831318e-05,
accuracy = 100.0%
Data type =  XOR
Activation function = relu
Optimizer type =  sgd
```

```
loss = 0.004305274256900546,
accuracy = 100.0%
Data type =  XOR
Activation function = lrelu
Optimizer type =  sgd
```

We can see that both ReLU and leaky ReLU have better converge speed.