Lab1: back-propagation Report

311605014 鄧奕辰

1. Introduction

In this project, we implement the neural network with forward and backward pass using two hidden layers. Using two kinds of data, xor and linear, to classify the input data. In this report, it will show how the neural network update the weights, and the comparison of different learning rates and different numbers of hidden units.

A. Dataset

- XOR

```
def generate_XOR_easy(self):
        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)
```

- Linear

```
def generate_linear(self, 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)
```

2. Experiment setups

A. Sigmoid functions and derivative of sigmoid function

Use sigmoid function as activation function.

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

B. Neural network

Neural network with two hidden layers, 2-bits input layer and 1-bit output layer. Each neuron has a weight and bias, and go through activation function. Derive as $Z = \sigma(wX + b)$.

Loss function will be MSE loss :

```
def mseloss(self, out, y):
    return np.mean((out - y)**2)
```

C. Backpropagation

The model initializes with random weight parameters.

```
# weights
self.w1 = np.random.randn(self.inputBits, self.neuronsInHidden)
self.w2 = np.random.randn(self.neuronsInHidden, self.neuronsInHidden)
self.w3 = np.random.randn(self.neuronsInHidden, self.outputBits)
self.b1 = np.zeros((self.neuronsInHidden, 1))
self.b2 = np.zeros((self.neuronsInHidden, 1))
self.b3 = np.zeros((self.outputBits, 1))
```

forwardPropagation function implement a very classic calculation, each neurons sum the previous output and pass through a activate function “sigmoid” and calculate the loss. The loss will be passed through back propagation by chain rule.

```
def backwardPropagation(self):
    self.dout = self.dloss(self.out, self.y)
    self.dz3 = np.multiply(self.dout, self.derivative_sigmoid(self.out))
    self.dw3 = np.dot(self.dz3, self.a2.T)
    self.db3 = np.sum(self.dz3, axis = 1, keepdims = True)

    self.da2 = np.dot(self.dz3.T, self.w3.T)
    self.dz2 = np.multiply(self.da2.T, self.derivative_sigmoid(self.a2))
    self.dw2 = np.dot(self.dz2, self.a1.T)
    self.db2 = np.sum(self.dz2, axis = 1, keepdims = True)

    self.da1 = np.dot(self.w2.T, self.dz2)
    self.dz1 = np.multiply(self.da1, self.derivative_sigmoid(self.a1))
    self.dw1 = np.dot(self.dz1, self.x)
    self.db1 = np.sum(self.dz1, axis = 1, keepdims = True)
```

```
def forwardPropagation(self):
    self.z1 = np.dot(self.w1.T, self.x.T) + self.b1
    self.a1 = self.sigmoid(self.z1)
    self.z2 = np.dot(self.w2, self.a1) + self.b2
    self.a2 = self.sigmoid(self.z2)
    self.z3 = np.dot(self.w3.T, self.a2) + self.b3
    self.out = self.sigmoid(self.z3)
```

After the backward propagation, the gradient of weight and bias are calculated, and all the weight will be updated by learning rate times gradient.

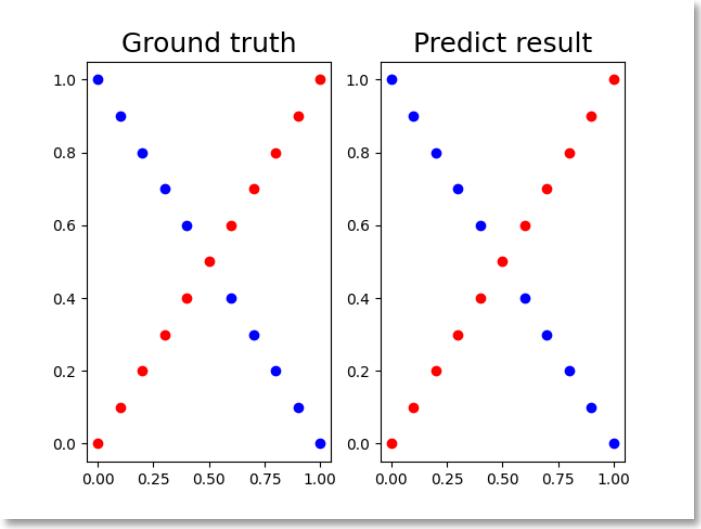
```
def updateParameters(self):
    self.w1 = self.w1 - self.lr * self.dw1.T
    self.w2 = self.w2 - self.lr * self.dw2.T
    self.w3 = self.w3 - self.lr * self.dw3.T

    self.b1 = self.b1 - self.lr * self.db1
    self.b2 = self.b2 - self.lr * self.db2
    self.b3 = self.b3 - self.lr * self.db3
```

3. Results of your testing

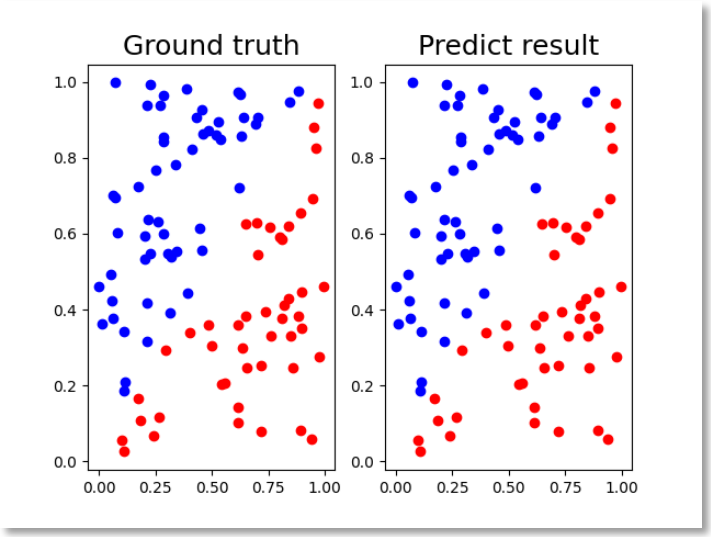
   A. Screenshot and comparison figure
   XOR train with 100000 epochs:

```
xor
epoch 10000, loss0.00021665157713895113
epoch 20000, loss7.525812193766898e-05
epoch 30000, loss4.325591439450523e-05
epoch 40000, loss2.9741813628708107e-05
epoch 50000, loss2.2422319553912733e-05
epoch 60000, loss1.787813601387215e-05
epoch 70000, loss1.4801616122696745e-05
epoch 80000, loss1.2589972319278109e-05
epoch 90000, loss1.092852007062138e-05
epoch 100000, loss9.637626247728192e-06
```



Linear train with 500000 epochs:

```
epoch 350000, loss6.190853688784203e-06
epoch 360000, loss6.0781703380809555e-06
epoch 370000, loss5.972913085206093e-06
epoch 380000, loss5.874522443639835e-06
epoch 390000, loss5.7824994436056335e-06
epoch 400000, loss5.69639799550097e-06
epoch 410000, loss5.61581840237115e-06
epoch 420000, loss5.540401826447606e-06
epoch 430000, loss5.469825551842524e-06
epoch 440000, loss5.403798914841438e-06
epoch 450000, loss5.342059796565648e-06
epoch 460000, loss5.284371591520759e-06
epoch 470000, loss5.2305205806044764e-06
epoch 480000, loss5.180313649355211e-06
epoch 490000, loss5.133576302159978e-06
epoch 500000, loss5.0901509312622685e-06
```



B. Show the accuracy of your prediction

XOR test:

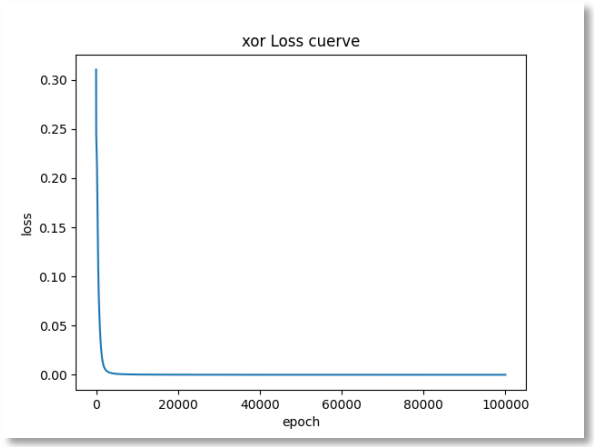
```
Iter1  | ground truth:0| pridiction:7e-05
Iter2  | ground truth:1| pridiction:1.0
Iter3  | ground truth:0| pridiction:6e-05
Iter4  | ground truth:1| pridiction:1.0
Iter5  | ground truth:0| pridiction:0.00014
Iter6  | ground truth:1| pridiction:1.0
Iter7  | ground truth:0| pridiction:0.00075
Iter8  | ground truth:1| pridiction:0.99999
Iter9  | ground truth:0| pridiction:0.00415
Iter10 | ground truth:1| pridiction:0.99241
Iter11 | ground truth:0| pridiction:0.00756
Iter12 | ground truth:0| pridiction:0.0047
Iter13 | ground truth:1| pridiction:0.99341
Iter14 | ground truth:0| pridiction:0.00188
Iter15 | ground truth:1| pridiction:1.0
Iter16 | ground truth:0| pridiction:0.00074
Iter17 | ground truth:1| pridiction:1.0
Iter18 | ground truth:0| pridiction:0.00033
Iter19 | ground truth:1| pridiction:1.0
Iter20 | ground truth:0| pridiction:0.00017
Iter21 | ground truth:1| pridiction:1.0
acc:100.0%
```

Linear test:

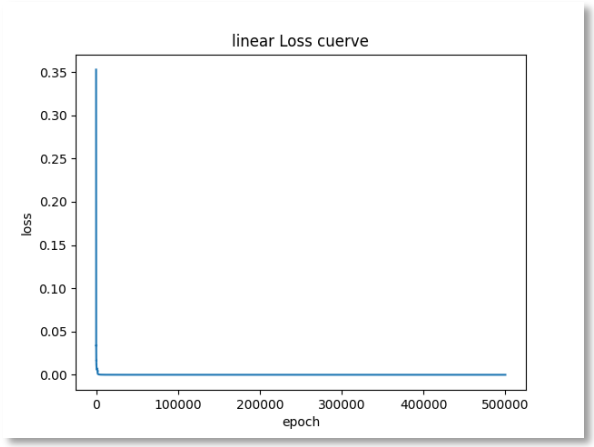
```
Iter80 | ground truth:1| pridiction:1.0
Iter81 | ground truth:0| pridiction:0.0
Iter82 | ground truth:0| pridiction:0.0
Iter83 | ground truth:1| pridiction:1.0
Iter84 | ground truth:1| pridiction:1.0
Iter85 | ground truth:0| pridiction:0.0
Iter86 | ground truth:0| pridiction:0.0
Iter87 | ground truth:1| pridiction:1.0
Iter88 | ground truth:1| pridiction:1.0
Iter89 | ground truth:0| pridiction:0.0
Iter90 | ground truth:1| pridiction:1.0
Iter91 | ground truth:0| pridiction:0.0
Iter92 | ground truth:1| pridiction:1.0
Iter93 | ground truth:1| pridiction:1.0
Iter94 | ground truth:1| pridiction:1.0
Iter95 | ground truth:1| pridiction:1.0
Iter96 | ground truth:1| pridiction:1.0
Iter97 | ground truth:1| pridiction:1.0
Iter98 | ground truth:0| pridiction:0.0
Iter99 | ground truth:0| pridiction:0.0
Iter100| ground truth:0| pridiction:0.0
acc:100.0%
```

C. Learning curve

XOR:



Linear:



4. Discussion

A. Try different learning rates

Using linear as example, I set learning rate = 0.1 and epoch = 15000. The testing accuracy achieve 100%.
Using linear as example, I set learning rate = 0.05 and epoch = 12000. The testing accuracy only achieve 97%.
Using linear as example, I set learning rate = 0.5 and epoch = 10000. The testing accuracy only achieve 98%, because of overshooting.
Using linear as example, I set learning rate = 0.5 and epoch = 5000. The testing accuracy achieve 100%.

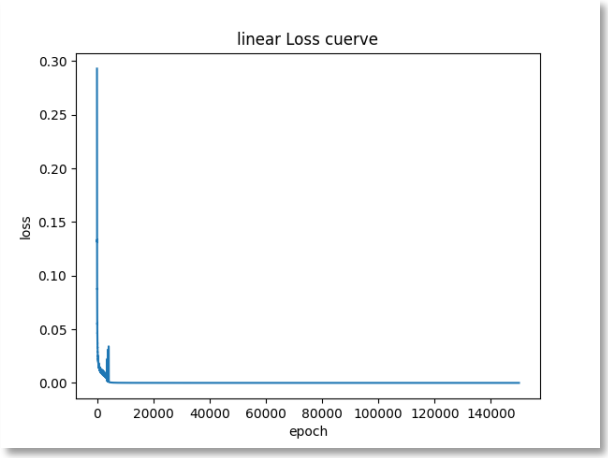
Based on the experiments conducted above, it was found that it is necessary to set the learning rate to an appropriate value and, if necessary, gradually reduce it to enable the model to converge to the global minimum.

B. Try different numbers of hidden units

For the pictures shown above, all the hidden layers were set to 4 units.

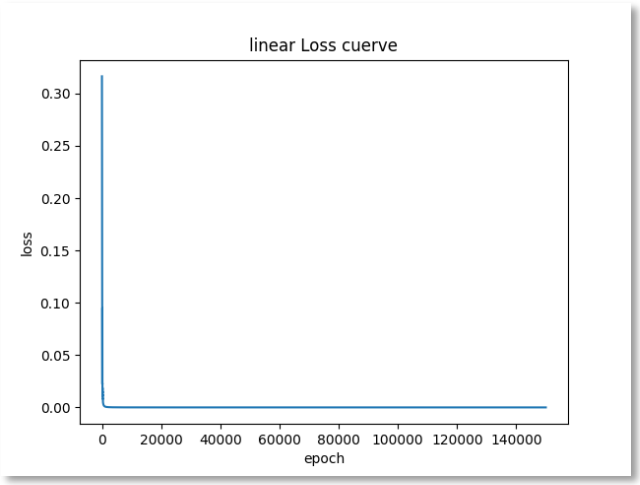
Using linear as example, I set two hidden layers with 10 units, learning rate = 0.1, epoch = 15000.

```
Iter95 | ground truth:1| pridiction:1.0
Iter96 | ground truth:1| pridiction:1.0
Iter97 | ground truth:1| pridiction:1.0
Iter98 | ground truth:0| pridiction:0.0
Iter99 | ground truth:0| pridiction:0.0
Iter100| ground truth:1| pridiction:1.0
acc:98.0%
```



Using linear as example, I set two hidden layers with 2 units, learning rate = 0.1, epoch = 15000.

```
Iter95 | ground truth:0| pridiction:2e-05
Iter96 | ground truth:1| pridiction:0.99998
Iter97 | ground truth:0| pridiction:2e-05
Iter98 | ground truth:1| pridiction:0.99993
Iter99 | ground truth:0| pridiction:6e-05
Iter100| ground truth:0| pridiction:2e-05
acc:99.0%
```



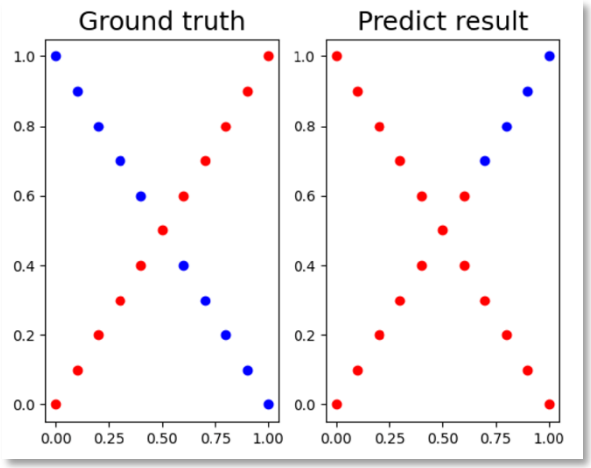
To summarize the above experiments, increasing the number of neurons in the hidden layer does not necessarily lead to better performance. In fact, it can increase the difficulty of convergence. However, when dealing with more complex features and tasks in harder datasets, increasing the distribution of neurons can result in better performance.

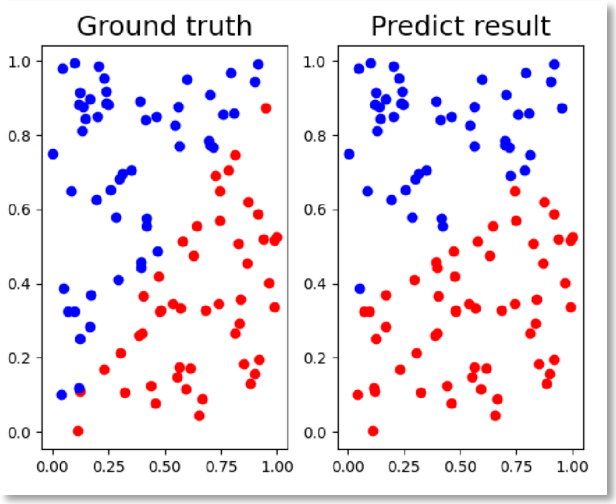
C. Try without activation functions

```
def forwardPropagation(self):
    self.z1 = np.dot(self.w1.T, self.x.T)
    self.z2 = np.dot(self.w2, self.z1)
    self.z3 = np.dot(self.w3.T, self.z2)
    self.out = self.z3
```

```
def backwardPropagation(self):
    self.dout = self.dloss(self.out, self.y)
    self.dz3 = self.dout
    self.dw3 = np.dot(self.dz3, self.a2.T)
    self.dz2 = np.multiply(self.da2.T, self.w3)
    self.dw2 = np.dot(self.dz2, self.a1.T)
    self.dz1 = np.multiply(self.da1, self.w2)
    self.dw1 = np.dot(self.dz1, self.x)
```

If I don’t use any activation function in a neural network, it would just become a linear regression model which can simplified as (X*W).
Moreover, the hidden layers would not learn any non-linear relationship in the data.
To summarize the XOR experiments without activation, it’s impossible to draw a single line to describe the relationship of XOR, therefore the result of XOR will have a bad accuracy. However, in the linear data which has a nearly linear relationship, therefore it has an ordinary performance.



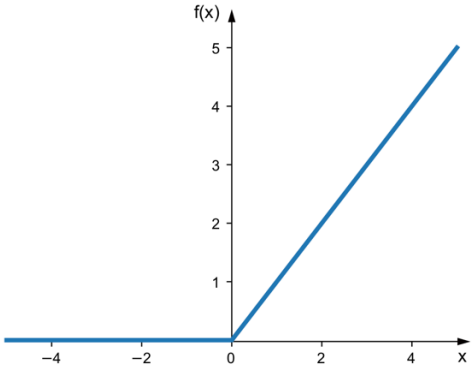


5. Extra

   A. Implement different optimizers
   B. Implement different activation functions

   Use ReLU activation function, shown as below. For the first and second activation function, I use ReLU activation function, then through sigmoid to output.

$$\mathrm{ReLU}(x) = \max(0, x)$$



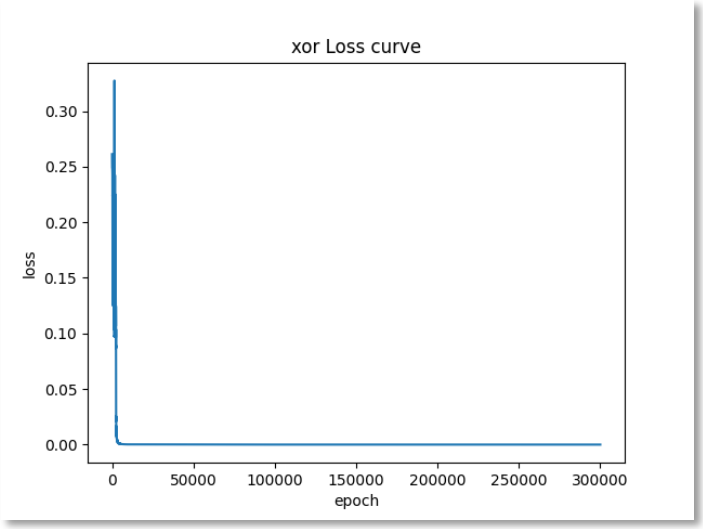

```
def forwardPropagation(self):
    self.z1 = np.dot(self.w1.T, self.x.T) + self.b1
    self.a1 = self.relu(self.z1)
    self.z2 = np.dot(self.w2, self.a1) + self.b2
    self.a2 = self.relu(self.z2)
    self.z3 = np.dot(self.w3.T, self.a2) + self.b3
    self.out = self.sigmoid(self.z3)
```

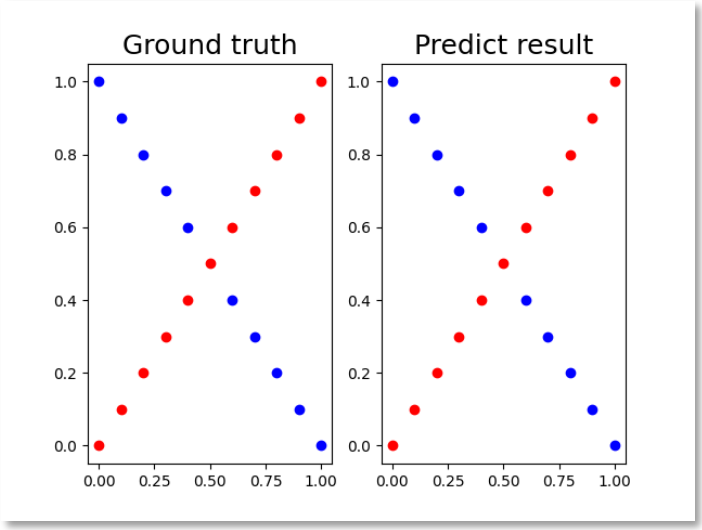
```python
def backwardPropagation(self):
    self.dout = self.dloss(self.out, self.y)
    self.dz3 = np.multiply(self.dout, self.derivative_sigmoid(self.out))
    self.dw3 = np.dot(self.dz3, self.a2.T)
    self.db3 = np.sum(self.dz3, axis = 1, keepdims = True)

    self.da2 = np.dot(self.dz3.T, self.w3.T)
    self.dz2 = np.multiply(self.da2.T, self.derivative_relu(self.a2))
    self.dw2 = np.dot(self.dz2, self.a1.T)
    self.db2 = np.sum(self.dz2, axis = 1, keepdims = True)

    self.da1 = np.dot(self.w2.T, self.dz2)
    self.dz1 = np.multiply(self.da1, self.derivative_relu(self.a1))
    self.dw1 = np.dot(self.dz1, self.x)
    self.db1 = np.sum(self.dz1, axis = 1, keepdims = True)
```

- XOR train with 30000 epoch:



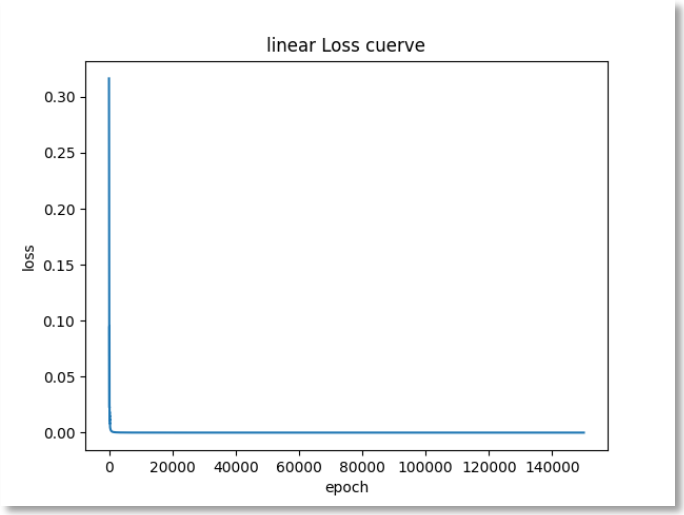


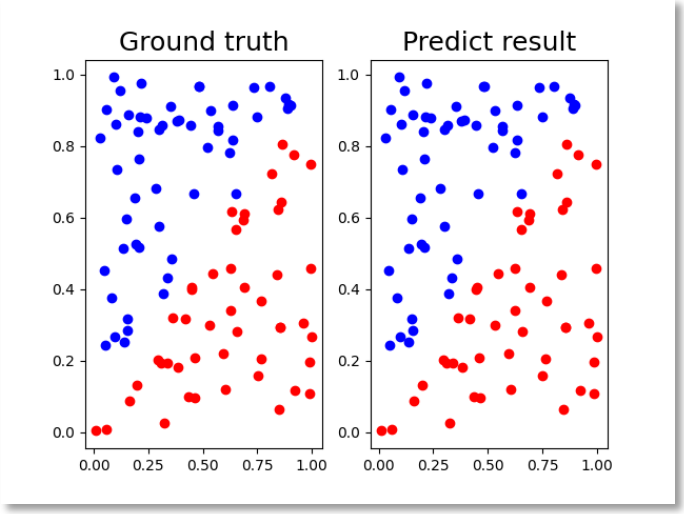
- XOR testing:

  The testing result shows that, relu activation with sigmoid activation still have a good performance.

```
Iter1  | ground truth:0| pridiction:0.00202
Iter2  | ground truth:1| pridiction:1.0
Iter3  | ground truth:0| pridiction:0.00202
Iter4  | ground truth:1| pridiction:1.0
Iter5  | ground truth:0| pridiction:0.00202
Iter6  | ground truth:1| pridiction:0.99999
Iter7  | ground truth:0| pridiction:0.00202
Iter8  | ground truth:1| pridiction:0.99999
Iter9  | ground truth:0| pridiction:0.00202
Iter10 | ground truth:1| pridiction:0.99789
Iter11 | ground truth:0| pridiction:0.00202
Iter12 | ground truth:0| pridiction:0.00202
Iter13 | ground truth:1| pridiction:0.99602
Iter14 | ground truth:0| pridiction:0.00202
Iter15 | ground truth:1| pridiction:1.0
Iter16 | ground truth:0| pridiction:0.00202
Iter17 | ground truth:1| pridiction:1.0
Iter18 | ground truth:0| pridiction:0.00202
Iter19 | ground truth:1| pridiction:1.0
Iter20 | ground truth:0| pridiction:0.00202
Iter21 | ground truth:1| pridiction:1.0
acc:100.0%
```

- Linear train with 25000 epoch:





- Linear test:
  The testing result shows that, relu activation with sigmoid activation still have a good performance.

```
Iter80 | ground truth:0| pridiction:1e-05
Iter81 | ground truth:0| pridiction:0.0
Iter82 | ground truth:0| pridiction:0.0
Iter83 | ground truth:0| pridiction:0.0
Iter84 | ground truth:0| pridiction:0.0
Iter85 | ground truth:1| pridiction:0.99993
Iter86 | ground truth:0| pridiction:0.0
Iter87 | ground truth:1| pridiction:0.99995
Iter88 | ground truth:0| pridiction:0.0
Iter89 | ground truth:1| pridiction:0.9997
Iter90 | ground truth:1| pridiction:0.99993
Iter91 | ground truth:1| pridiction:0.99993
Iter92 | ground truth:0| pridiction:0.0
Iter93 | ground truth:0| pridiction:0.0
Iter94 | ground truth:1| pridiction:0.99993
Iter95 | ground truth:0| pridiction:0.0
Iter96 | ground truth:1| pridiction:0.99993
Iter97 | ground truth:0| pridiction:1e-05
Iter98 | ground truth:0| pridiction:0.0
Iter99 | ground truth:1| pridiction:0.99997
Iter100| ground truth:1| pridiction:0.99993
acc:100.0%
```

C. Implement convolutional layers