Lab1 : back-propagation

Lab Objective:

In this lab, you will need to understand and implement simple neural networks with forwarding pass and backpropagation using two hidden layers. Notice that you can only use **Numpy** and the python standard libraries, any other frameworks (ex : Tensorflow、PyTorch) are not allowed in this lab.

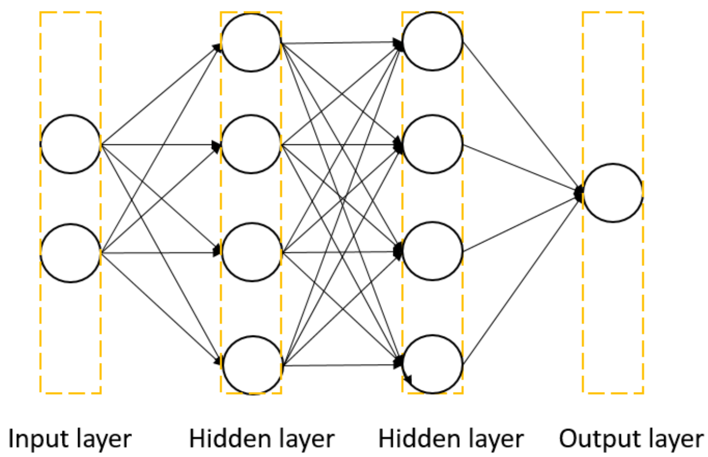


Figure 1. A two-layer neural network

Important Date:

1. Experiment Report Submission Deadline: 7/20 (Tue.) 12:00 p.m.
2. Demo date: 7/20 (Tue.)

Turn in:

1. Experiment Report (.pdf)
2. Source code

Notice: zip all files in one file and name it like「DLP_LAB1_your studentID_name.zip」, ex:「DLP_LAB1_309551027_李美慧.zip」

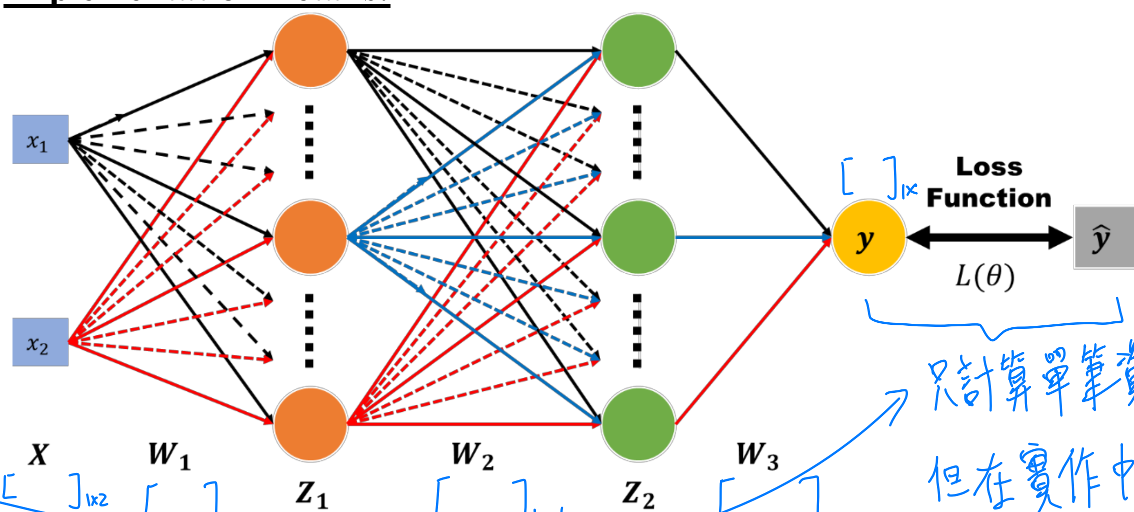
Requirements:

1. Implement simple neural networks with two hidden layers.
2. You must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
3. Plot your comparison figure that show the predicted results and the ground-truth.

suppose amount of hidden units in hidden layer 1 and 2 are h_1 and h_2 respectively

Implementation Details:

一次處理單筆資料



$[\ ]_{1\times 2}$ $[\ ]_{2\times h_1}$ $[\ ]_{h_1\times h_2}$ $[\ ]_{h_2\times 1}$ $[\ ]_{1\times}$

只計算單筆資料的error

但在實作中,我會一次處理多筆資料

Figure 2. Forward pass

- In the figure 2, we use the following definitions for the notations:
  1. x_1, x_2 : *nerual network inputs*
  2. X : $[x_1, x_2]$
  3. y : *nerual network outputs*
  4. $\hat{y}$: *ground truth*
  5. L(θ) : *loss function*
  6. W_1, W_2, W_3 : *weight matrix of network layers*

- Here are the computations represented:

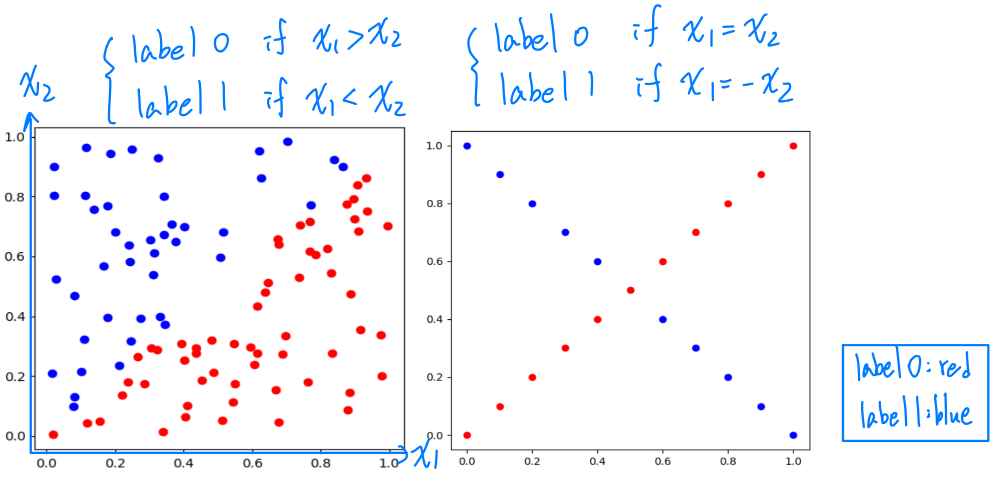
  $\boldsymbol{Z_1} = \boldsymbol{\sigma}(\boldsymbol{X}W_1)$ $\quad$ $\boldsymbol{Z_2} = \boldsymbol{\sigma}(\boldsymbol{Z_1}W_2)$ $\quad$ $\boldsymbol{y} = \boldsymbol{\sigma}(\boldsymbol{Z_2}W_3)$

- In the equations, the $\boldsymbol{\sigma}$ is sigmoid function that refers to the special case of the **logistic** function and defined by the formula:

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

- **Input / Test:**

  The inputs are two kinds which showing at below.



You need to use the following generate functions to create your inputs x, y.

```
def generate_linear(n=100):
    import numpy as np
    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)
```

$x=y$

```
def generate_XOR_easy():
    import numpy as np
    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)
```

Function usage

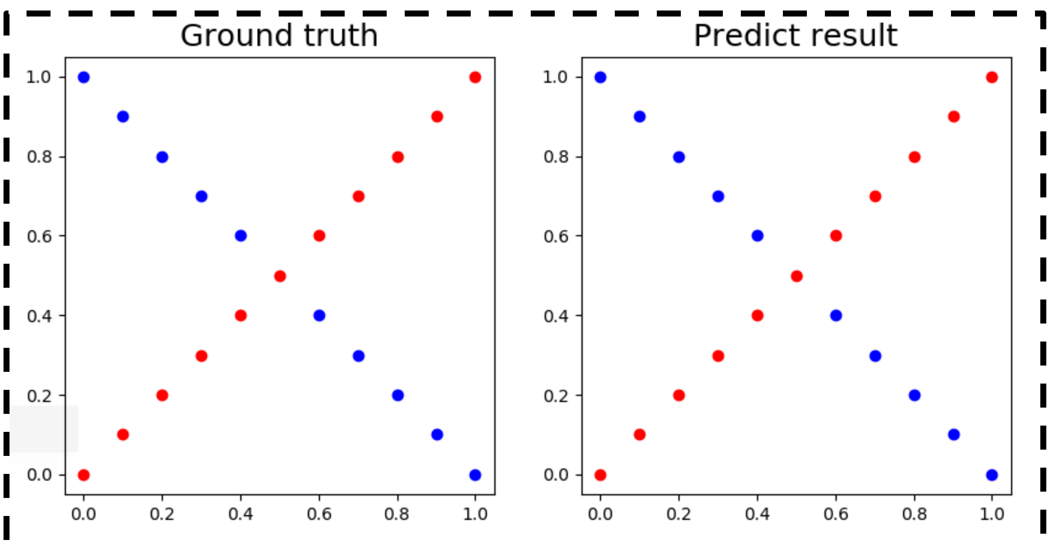
```
x, y = generate_linear(n=100)
x, y = generate_XOR_easy()
```

In the training, you need to print the loss values; In the testing, you need to show your predictions as shown below.

```
epoch 10000 loss : 0.16234523253277644
epoch 15000 loss : 0.2524336634177614
epoch 20000 loss : 0.1590783047540092
epoch 25000 loss : 0.22099447030234853
epoch 30000 loss : 0.3292173477217561
epoch 35000 loss : 0.40406233282426085
epoch 40000 loss : 0.43052897480298924
epoch 45000 loss : 0.420752573558660 5
epoch 50000 loss : 0.3934759509342479
epoch 55000 loss : 0.3615008372106921
epoch 60000 loss : 0.33077879872648525
epoch 65000 loss : 0.30333537090819584
epoch 70000 loss : 0.2794858089741792
epoch 75000 loss : 0.25892812312991587
epoch 80000 loss : 0.24119780823897027
epoch 85000 loss : 0.22583656353511342
epoch 90000 loss : 0.21244497028971704
epoch 95000 loss : 0.2006912468389013
```

```
[[0.01025062]
 [0.99730607]
 [0.02141321]
 [0.99722154]
 [0.03578171]
 [0.99701922]
 [0.04397049]
 [0.99574117]
 [0.04162245]
 [0.92902792]
 [0.03348791]
 [0.02511045]
 [0.94093942]
 [0.01870069]
 [0.99622948]
 [0.01431959]
 [0.99434455]
 [0.01143039]
 [0.98992477]
 [0.00952752]
 [0.98385905]]
```

Visualize the predictions and ground truth at the end of the training process.
The comparison figure should look like the example below.



You can refer to the following visualization code

x: inputs (2-dimensional array)

y: ground truth label (1-dimensional array)

pred_y: outputs of neural network (1-dimensional array)

```python
def show_result(x, y, pred_y):
    import matplotlib.pyplot as plt
    plt.subplot(1,2,1)
    plt.title('Ground truth', fontsize=18)
    for i in range(x.shape[0]):
        if y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.subplot(1,2,2)
    plt.title('Predict result', fontsize=18)
    for i in range(x.shape[0]):
        if pred_y[i] == 0:
            plt.plot(x[i][0], x[i][1], 'ro')
        else:
            plt.plot(x[i][0], x[i][1], 'bo')

    plt.show()
```

- **Sigmoid functions:**
  1. A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. It is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point. In general, a sigmoid function is monotonic, and has a first derivative which is bell shaped.
  2. (hint) You may write the function like this:

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

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

  3. (hint) The derivative of sigmoid function

```python
def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```

$\sigma'(x) = \frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$

- **Back Propagation (Gradient computation)**
  Backpropagation is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network. Backpropagation is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to

iteratively compute gradients for each layer. The backpropagation learning algorithm can be divided into two parts; **propagation** and **weight update**.

Part 1: Propagation

Each propagation involves the following steps:

1. Propagation forward through the network to generate the output value
2. Calculation of the cost $L(\theta)$ (error term)
3. Propagation of the output activations back through the network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

Part 2: Weight update

For each weight-synapse follow the below steps:

1. Multiply its output delta and input activation to get the gradient of the weight.
2. Subtract a ratio (percentage) of the gradient from the weight.
3. This ratio (percentage) influences the speed and quality of learning; it is called the **learning rate**. The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

Repeat part. 1 and 2 until the performance of the network is satisfactory.

Pseudocode:

```
initialize network weights (often small random values)
do
   forEach training example named ex
      prediction = neural-net-output(network, ex)  // forward pass
      actual = teacher-output(ex)
      compute error (prediction - actual) at the output units
      compute Δw_h for all weights from hidden layer to output layer  // backward pass
      compute Δw_i for all weights from input layer to hidden layer   // backward pass continued
      update network weights // input layer not modified by error estimate
until all examples classified correctly or another stopping criterion satisfied
return the network
```

Report Spec

1. Introduction (20%)
2. Experiment setups (30%):
   A. Sigmoid functions
   B. Neural network
   C. Backpropagation
3. Results of your testing (20%)
   A. Screenshot and comparison figure
   B. Show the accuracy of your prediction
   C. Learning curve (loss, epoch curve)
   D. anything you want to present
4. Discussion (30%)
   A. Try different learning rates
   B. Try different numbers of hidden units
   C. Try without activation functions
   D. Anything you want to share

Score:
60% demo score (experimental results & questions) + 40% report
For experimental results, you have to achieve at least 90% of accuracy to get the demo score.
If the zip file name or the report spec have format error, you will be punished (-5)

Reference:

1. Logical regression:
   http://www.bogotobogo.com/python/scikit-learn/logistic_regression.php

2. Python tutorial:
   https://docs.python.org/3/tutorial/

3. Numpy tutorial:
   https://www.tutorialspoint.com/numpy/index.htm

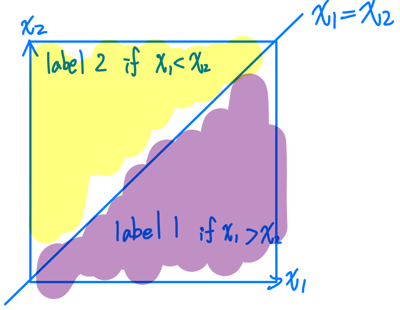
4. Python Standard Library:
   https://docs.python.org/3/library/index.html

5. http://speech.ee.ntu.edu.tw/~tlkagk/courses/ML_2016/Lecture/BP.pdf
6. https://en.wikipedia.org/wiki/Sigmoid_function
7. https://en.wikipedia.org/wiki/Backpropagation

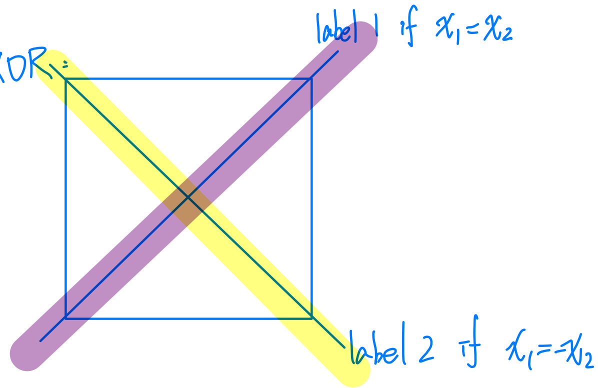
1. Introduction (20%)

Ground true data generation

linear :



XOR :



Construct neural network model

suppose the amount of hidden units for first and second layer are h1 and h2 respectively
original version only deals with one data for each iteration, but in fact we can deal with multiple (batch size) data for each iteration

Implementation Details:

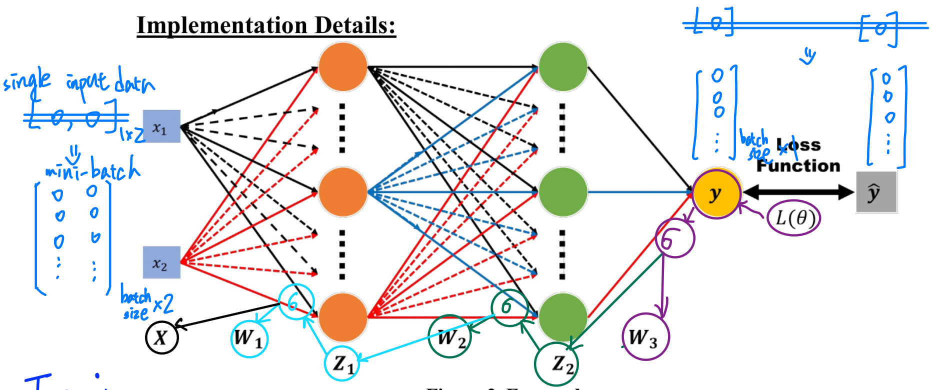


Figure 2. Forward pass

Train

randomly initialize the network weights (no bias)

W_1, W_2, W_3

while not converge (for each epoch check whether loss is smaller than epsilon or not)

$$Loss = \frac{-1}{batchsize} \sum_{i \in minibatch} \left(\hat{y}_i \log y_i + (1-\hat{y}_i) \log (1-y_i) \right)$$

the design of loss function is related to the problem and network design. it will be explained precisely in 2.B.

for each mini-batch

forward

$$\sigma(XW_1) = Z_1$$
$$\sigma(Z_1W_2) = Z_2 \quad , \text{ where } \sigma(x) = \frac{1}{1+e^{-x}} \quad , \quad \sigma'(x) = \sigma(x)(1-\sigma(x))$$
$$\sigma(Z_2W_3) = y$$

we want to adjust model weights W1, W2, and W3 to lower down loss function, so we compute the gradient which represents the steepest direction to update them the computation details will be discussed in 2.C.

backward

matrix product

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial W_3}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial Z_2} @ \frac{\partial Z_2}{\partial (Z_1W_2)} @ \frac{\partial (Z_1W_2)}{\partial W_2}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial Z_2} @ \frac{\partial Z_2}{\partial (Z_1W_2)} @ \frac{\partial (Z_1W_2)}{\partial Z_1} @ \frac{\partial Z_1}{\partial (XW_1)} @ \frac{\partial (XW_1)}{\partial W_1}$$

update network weights

$$W_1 = W_1 - \text{learning rate} \cdot \frac{\partial L}{\partial W_1}$$
$$W_2 = W_2 - \text{learning rate} \cdot \frac{\partial L}{\partial W_2}$$
$$W_3 = W_3 - \text{learning rate} \cdot \frac{\partial L}{\partial W_3}$$

* W_1, W_2, W_3 不具相關性,
只有 Z_1, Z_2, y 才具相關性!!!

Test

forward the trained network and output the predicted y
print the accuracy

2. Experiment setups (30%):

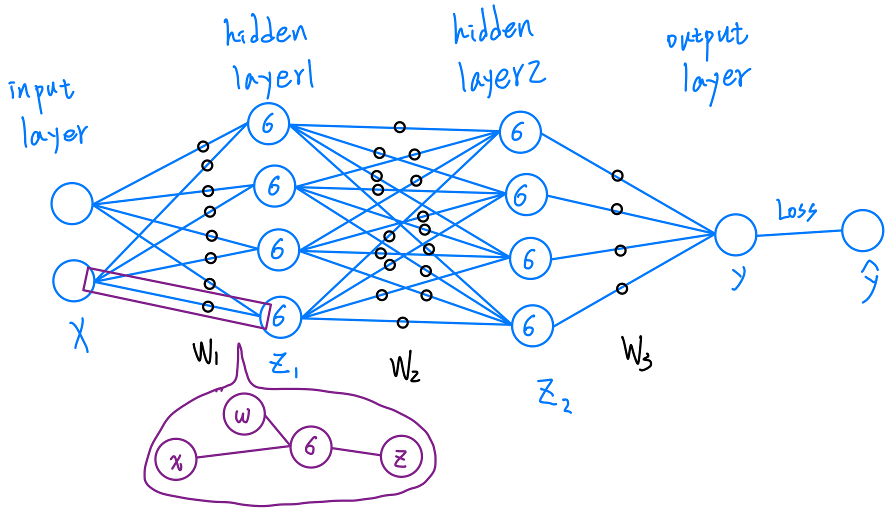
A. Sigmoid functions

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

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

```python
def sigmoid(M):
    return 1.0/(1.0+np.exp(-M))

def derivative_sigmoid(M):
    return sigmoid(M)*(1-sigmoid(M))
```

B. Neural network



amount of hidden units for first and second layer are h1=10 and h2=10 respectively

learning rate is 0.3

epsilon is 0.01

model weights W1, W2, W3 are randomly initialized with size (2,h1), (h1,h2), (h2,1) respectively

```python
#initialize model parameter
nHiddenUnits=(10,10) #amount of hidden units for each layer
learningRate = 0.3 #learning rate
epsilon = 0.01 #to judge converge or not

def __init__(self,nHiddenUnits,learningRate,epsilon):
    (h1,h2) = nHiddenUnits
    self.lr = learningRate
    self.eps = epsilon
    self.W1 = np.random.randn(2,h1)
    self.W2 = np.random.randn(h1,h2)
    self.W3 = np.random.randn(h2,1)
```

the network forward parameters like this:

$$\sigma(XW_1) = Z_1$$
$$\sigma(Z_1W_2) = Z_2$$
$$\sigma(Z_2W_3) = y$$

```python
def forward(self,bX):
    self.inputs = bX
    self.Z1 = sigmoid(self.inputs@self.W1)
    self.Z2 = sigmoid(self.Z1@self.W2)
    pred_y = sigmoid(self.Z2@self.W3)
    return pred_y
```

the loss function is defined like this:

since it's a binary classification problem and we embed the label using one hot encoding method, we can view it as logistic regression problem which use sigmoid function as activation function and use binary cross entropy as loss function

$$Loss = \frac{-1}{batchsize} \sum_{i \in minibatch} \left(\hat{y}_i \log y_i + (1-\hat{y}_i) \log(1-y_i) \right)$$

y has been activated by sigmoid function

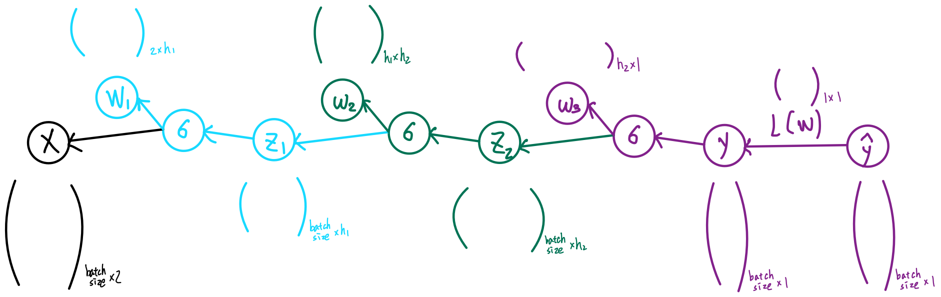
```python
def loss(self,gt_y,pred_y):
    batchsize = gt_y.shape[0]
    return (-1/batchsize) * np.sum(
        gt_y*np.log(gt_y+self.eps)
        +(1-pred_y)*np.log(1-pred_y+self.eps))
```

C. Backpropogation

here shows the detailed backward computation using chain rule:

$\odot$: Hardamard product

@: standard matrix product



$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial W_3}$$

it's actually a diagonal matrix (only the diagonal has value, and the other elements are all zero), so the standard matrix product of two matrices are equal to the Hardamard product (element-wise multiplication) of two matrices

$\sigma(Z_2W_3)_k(1-\sigma(Z_2W_3)_k)$

$$= \left\{ \left[\frac{-1}{batchsize} \begin{matrix} \hat{y}_1\cdot\frac{1}{y_1} + (1-\hat{y}_1)\cdot\frac{1}{1-y_1}\cdot(-1) \\ \hat{y}_2\cdot\frac{1}{y_2} + (1-\hat{y}_2)\cdot\frac{1}{1-y_2}\cdot(-1) \\ \vdots \end{matrix} \right]^T_{\substack{batch\ size\times 1 \\ 1\times batch\ size}} @ \begin{bmatrix} \frac{\partial y_1}{\partial (Z_2W_3)_1} & \frac{\partial y_1}{\partial (Z_2W_3)_2} & \frac{\partial y_1}{\partial (Z_2W_3)_3} & \cdots \\ \frac{\partial y_2}{\partial (Z_2W_3)_1} & \frac{\partial y_2}{\partial (Z_2W_3)_2} & \frac{\partial y_2}{\partial (Z_2W_3)_3} & \cdots \end{bmatrix}_{batch\ size\times batch\ size} @ Z_{2\ batch\ size\times h_2} \right\}^T_{\substack{1\times h_2 \\ h_2\times 1}}$$

$$= \left\{ \left\{ \left[\frac{-1}{batchsize} \begin{matrix} \hat{y}_1\cdot\frac{1}{y_1} + (1-\hat{y}_1)\cdot\frac{1}{1-y_1}\cdot(-1) \\ \hat{y}_2\cdot\frac{1}{y_2} + (1-\hat{y}_2)\cdot\frac{1}{1-y_2}\cdot(-1) \\ \vdots \end{matrix} \right]_{batch\ size\times 1} \odot \begin{bmatrix} \partial(Z_2W_3)_1(1-\sigma(Z_2W_3)_1) \\ \partial(Z_2W_3)_2(1-\sigma(Z_2W_3)_2) \\ \vdots \end{bmatrix}_{batch\ size\times 1} \right\}^T_{\substack{batch\ size\times 1 \\ 1\times batch\ size}} @ Z_{2\ batch\ size\times h_2} \right\}^T_{\substack{1\times h_2 \\ h_2\times 1}}$$

has been computed

same technique (replace standard matrix product with Hardamard product)

$$\frac{\partial L}{\partial W_2} = \left\{ \left\{ \left[\frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} \right]_{batch\ size\times 1} @ \frac{\partial (Z_2W_3)}{\partial Z_2}{}^T_{\substack{h_2\times 1 \\ 1\times h_2}} @ \frac{\partial Z_2}{\partial (Z_1W_2)} \right\}^T_{\substack{batch\ size\times h_2 \\ h_2\times batch\ size}} @ \frac{\partial (Z_1W_2)}{\partial W_2}{}_{batch\ size\times h_1} \right\}^T_{\substack{h_2\times h_1 \\ h_1\times h_2}}$$

$$= \left\{ \left\{ \left[\frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} \right]_{batch\ size\times 1} @ W_3{}^T_{\substack{h_2\times 1 \\ 1\times h_2}} \odot \begin{bmatrix} \partial(Z_1W_2)_{11}(1-\sigma(Z_1W_2)_{11}) & \partial(Z_1W_2)_{12}(1-\sigma(Z_1W_2)_{12}) & \cdots \\ \partial(Z_1W_2)_{21}(1-\sigma(Z_1W_2)_{21}) & \partial(Z_1W_2)_{22}(1-\sigma(Z_1W_2)_{22}) & \cdots \\ \partial(Z_1W_2)_{31}(1-\sigma(Z_1W_2)_{31}) & \partial(Z_1W_2)_{32}(1-\sigma(Z_1W_2)_{32}) & \cdots \\ \vdots & \vdots \end{bmatrix}_{batch\ size\times h_2} \right\}^T_{\substack{batch\ size\times h_2 \\ h_2\times batch\ size}} @ Z_{1\ batch\ size\times h_1} \right\}^T_{\substack{h_2\times h_1 \\ h_1\times h_2}}$$

has been computed

same technique (replace standard matrix product with Hardamard product)

$$\frac{\partial L}{\partial W_1} = \left\{ \left\{ \left[\frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial Z_2} @ \frac{\partial Z_2}{\partial (Z_1W_2)} \right]_{batch\ size\times h_2} @ \frac{\partial (Z_1W_2)}{\partial Z_1}{}_{\substack{h_1\times h_2 \\ h_2\times h_1}} @ \frac{\partial Z_1}{\partial (XW_1)} \right\}^T_{\substack{batch\ size\times h_1 \\ h_1\times batch\ size}} @ \frac{\partial (XW_1)}{\partial W_1}{}_{batch\ size\times 2} \right\}^T_{\substack{h_1\times 2 \\ 2\times h_1}}$$

$$= \left\{ \left\{ \left[\frac{\partial L}{\partial y} @ \frac{\partial y}{\partial (Z_2W_3)} @ \frac{\partial (Z_2W_3)}{\partial Z_2} @ \frac{\partial Z_2}{\partial (Z_1W_2)} \right]_{batch\ size\times h_2} @ W_2{}^T_{\substack{h_1\times h_2 \\ h_2\times h_1}} @ \begin{bmatrix} \partial(XW_1)_{11}(1-\sigma(XW_1)_{11}) & \partial(XW_1)_{12}(1-\sigma(XW_1)_{12}) & \cdots \\ \partial(XW_1)_{21}(1-\sigma(XW_1)_{21}) & \partial(XW_1)_{22}(1-\sigma(XW_1)_{22}) & \cdots \\ \partial(XW_1)_{31}(1-\sigma(XW_1)_{31}) & \partial(XW_1)_{32}(1-\sigma(XW_1)_{32}) & \cdots \\ \vdots & \vdots \end{bmatrix}_{batch\ size\times h_1} \right\}^T_{\substack{batch\ size\times h_1 \\ h_1\times batch\ size}} @ X_{batch\ size\times 2} \right\}^T_{\substack{h_1\times 2 \\ 2\times h_1}}$$

$W_1 = W_1 - \text{learning rate} \cdot \frac{\partial L}{\partial W_1}$

$W_2 = W_2 - \text{learning rate} \cdot \frac{\partial L}{\partial W_2}$

$W_3 = W_3 - \text{learning rate} \cdot \frac{\partial L}{\partial W_3}$

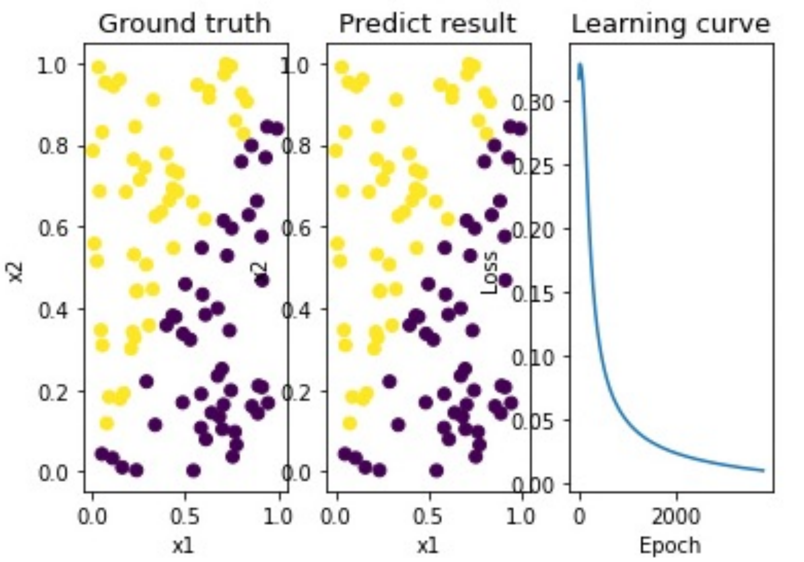
```
def backward(self,gt_y,pred_y):
    #backward propogation
    #dL/d{W3} = dL/dy * dy/d{Z2W3} * d{Z2W3}/d{W3}
    batchsize = gt_y.shape[0]

    grad_L_y = (-1/batchsize) * (gt_y/pred_y - (1-gt_y)/(1-pred_y))
    grad_L_Z2W3 = grad_L_y * derivative_sigmoid(self.Z2@self.W3)
    grad_L_W3 = (grad_L_Z2W3.T @ self.Z2).T

    #dL/d{W2} = dL/d{Z2W3} * d{Z2W3}/d{Z2} * d{Z2}/d{Z1W2} * d{Z1W2}/d{W2}
    grad_L_Z2 = grad_L_Z2W3 @ self.W3.T
    grad_L_Z1W2 = grad_L_Z2 * derivative_sigmoid(self.Z1@self.W2)
    grad_L_W2 = (grad_L_Z1W2.T @ self.Z1).T

    #dL/d{W1} = dL/d{Z1W2} * d{Z1W2}/d{Z1} * d{Z1}/d{XW1} * d{XW1}/d{W1}
    grad_L_Z1 = grad_L_Z1W2 @ self.W2.T
    grad_L_XW1 = grad_L_Z1 * derivative_sigmoid(self.inputs@self.W1)
    grad_L_W1 = (grad_L_XW1.T @ self.inputs).T

    #update model weights
    self.W1 = self.W1 - self.lr*grad_L_W1
    self.W2 = self.W2 - self.lr*grad_L_W2
    self.W3 = self.W3 - self.lr*grad_L_W3
```

3. Result of your testing (20%)

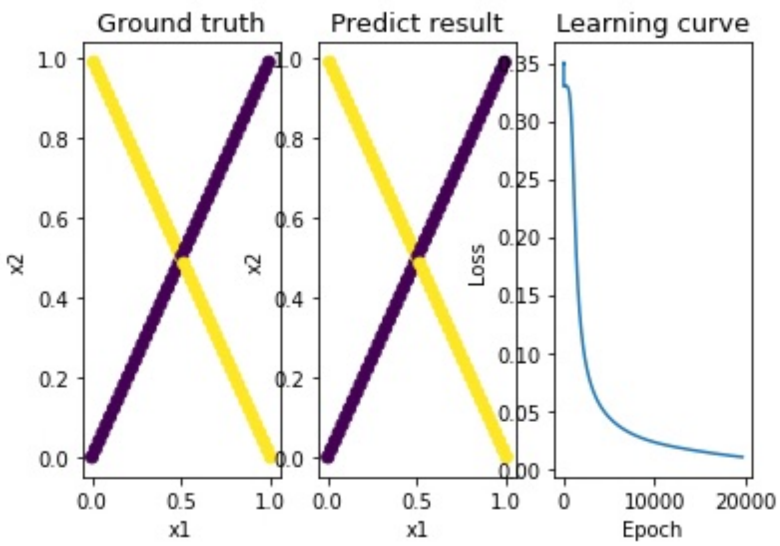
A. Screenshot and comparison figure

B. Show the accuracy of your prediction

C. Learning curve (loss, epoch curve)




```
=====data type : linear=====
training ... epoch:500, loss:0.09334, acc:1.00
training ... epoch:1000, loss:0.04846, acc:1.00
training ... epoch:1500, loss:0.03231, acc:1.00
training ... epoch:2000, loss:0.02372, acc:1.00
training ... epoch:2500, loss:0.01828, acc:1.00
training ... epoch:3000, loss:0.01443, acc:1.00
training ... epoch:3500, loss:0.01148, acc:1.00
testing ... accuracy:1.0
```




```
=====data type : XOR=====
training ... epoch:500, loss:0.32837, acc:0.80
training ... epoch:1000, loss:0.27320, acc:0.87
training ... epoch:1500, loss:0.16714, acc:0.92
training ... epoch:2000, loss:0.11478, acc:0.93
training ... epoch:2500, loss:0.08860, acc:0.95
training ... epoch:3000, loss:0.07297, acc:0.95
training ... epoch:3500, loss:0.06246, acc:0.96
training ... epoch:4000, loss:0.05483, acc:0.96
training ... epoch:4500, loss:0.04899, acc:0.97
training ... epoch:5000, loss:0.04435, acc:0.97
training ... epoch:5500, loss:0.04054, acc:0.97
training ... epoch:6000, loss:0.03734, acc:0.97
training ... epoch:6500, loss:0.03460, acc:0.97
training ... epoch:7000, loss:0.03223, acc:0.97
training ... epoch:7500, loss:0.03015, acc:0.97
training ... epoch:8000, loss:0.02833, acc:0.97
training ... epoch:8500, loss:0.02671, acc:0.97
training ... epoch:9000, loss:0.02527, acc:0.98
training ... epoch:9500, loss:0.02398, acc:0.98
⋮
training ... epoch:16000, loss:0.01376, acc:0.99
training ... epoch:16500, loss:0.01319, acc:0.99
training ... epoch:17000, loss:0.01264, acc:0.99
training ... epoch:17500, loss:0.01211, acc:0.99
training ... epoch:18000, loss:0.01160, acc:0.99
training ... epoch:18500, loss:0.01111, acc:0.99
training ... epoch:19000, loss:0.01064, acc:0.99
training ... epoch:19500, loss:0.01020, acc:0.99
testing ... accuracy:0.99
```

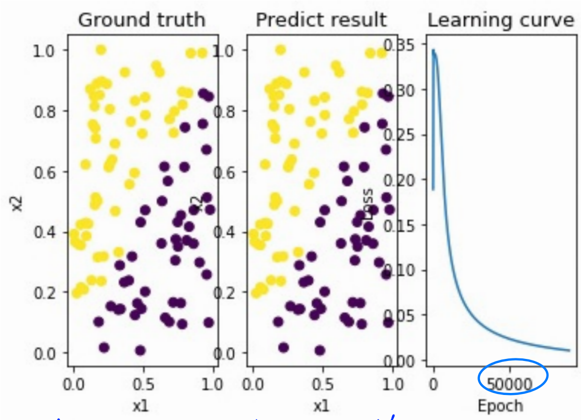
D. anything you want to present

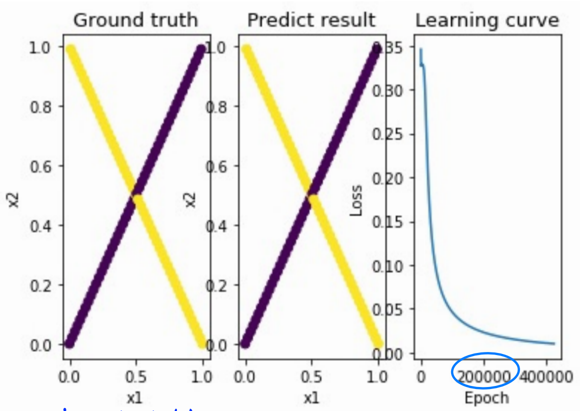
- in the backward propogation, because some matrices are actually diagonal matrices, the result of multiplying the diagonal matrices with other matrix is actually the same as directly multiplying two matrix's elements one by one. hence, in the implementation, I just use * instead of @.
- notice that the model weight of many neural network graphs are represented by lines, but the weights are actually also nodes. hence when we calculate backward propogation by hand, we need to draw the weights with nodes instead of lines, otherwise we can't know exactly where the chain rule does.

4. Discussion (30%)

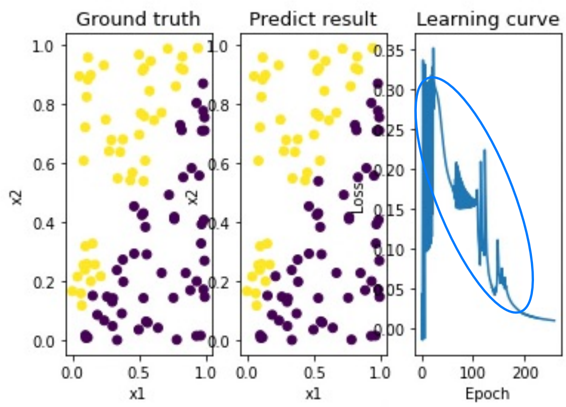
A. Try different learning rates

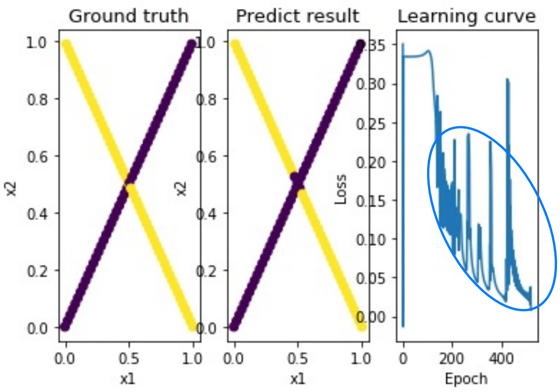
learning rate = 0.01, the network requires more epochs to converge





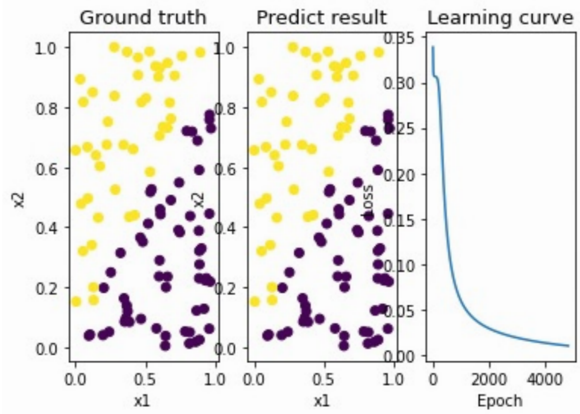
learning rate = 10, the network's convergence process is not stable

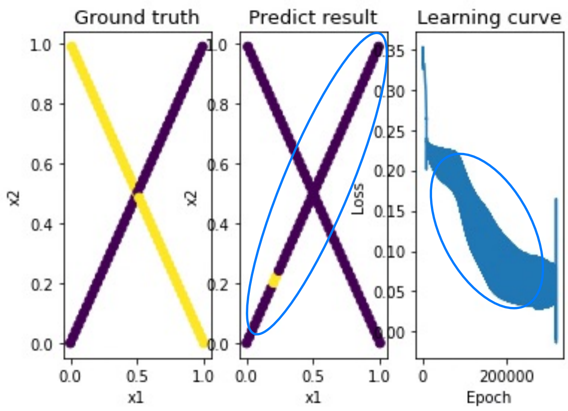




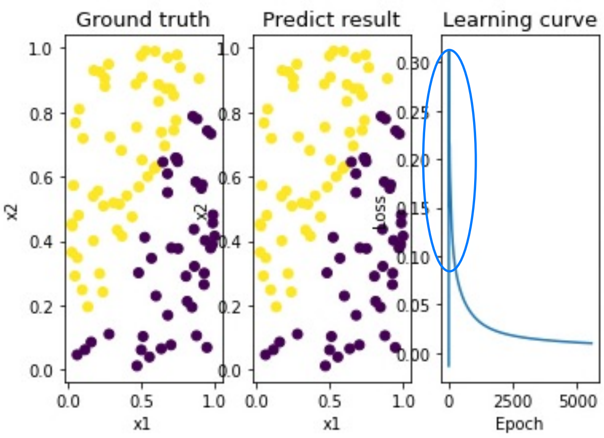
B. Try different numbers of hidden units

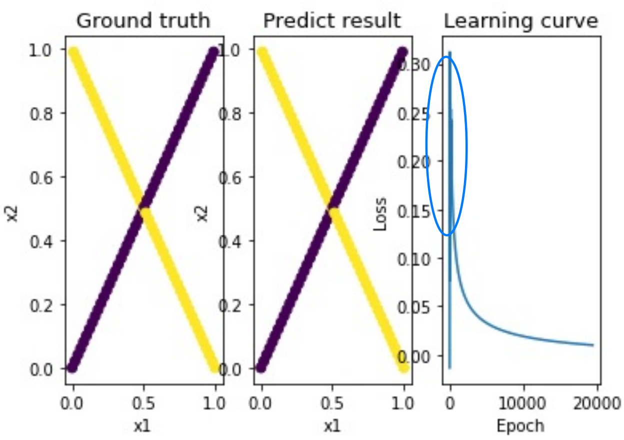
of hidden units for first and second hidden layer = (3,3) , the network is not stable and may not predict well



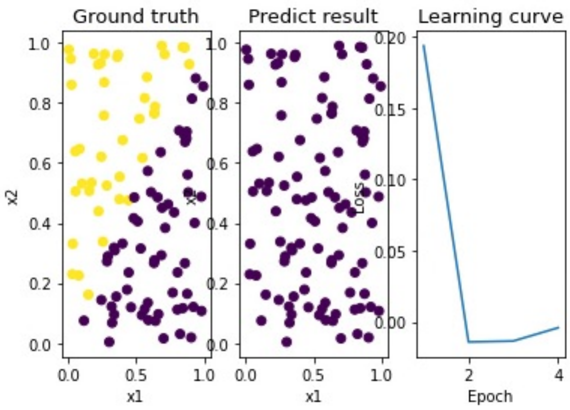


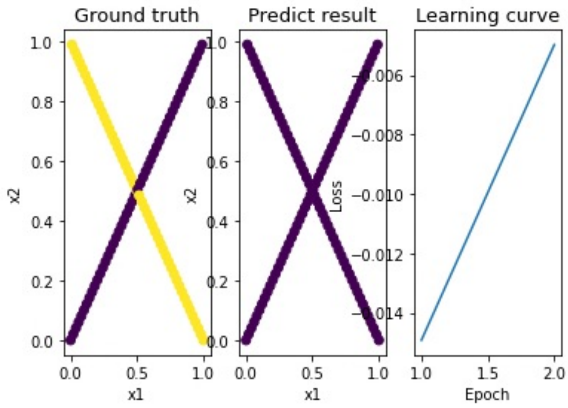
# of hidden units for first and second hidden layer = (70,70) , the loss decrease fast in the beginning





# of hidden units for first and second hidden layer = (1000,1000) , the network predict nothing





C. Try without activation function

the neural network can only fit the linear regression problem, so the training process won't converge

```
training ... epoch:500, loss:-34363.45243, acc:0.50
training ... epoch:1000, loss:-104405.06171, acc:0.50
training ... epoch:1500, loss:-201080.83092, acc:0.50
training ... epoch:2000, loss:-320438.23154, acc:0.50
training ... epoch:2500, loss:-460040.43090, acc:0.50
training ... epoch:3000, loss:-618173.92873, acc:0.50
training ... epoch:3500, loss:-793542.07872, acc:0.50
training ... epoch:4000, loss:-985115.95048, acc:0.50
training ... epoch:4500, loss:-1192051.30267, acc:0.50
training ... epoch:5000, loss:-1413637.99221, acc:0.50
training ... epoch:5500, loss:-1649267.06546, acc:0.50
```