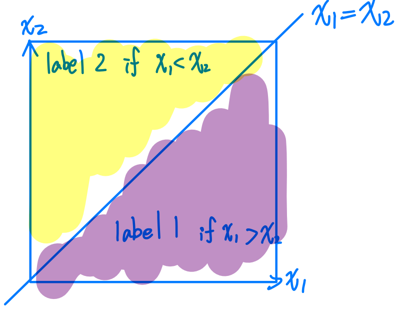
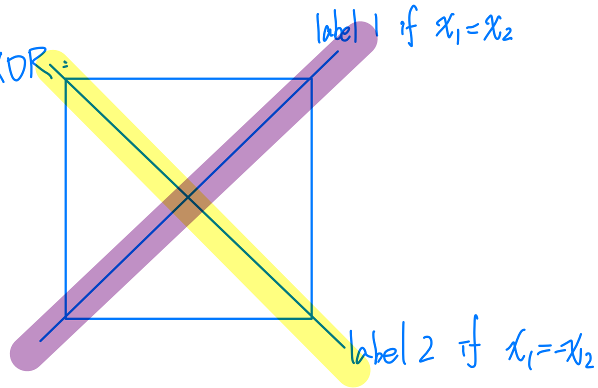
# 1. Introduction (20%)

## 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

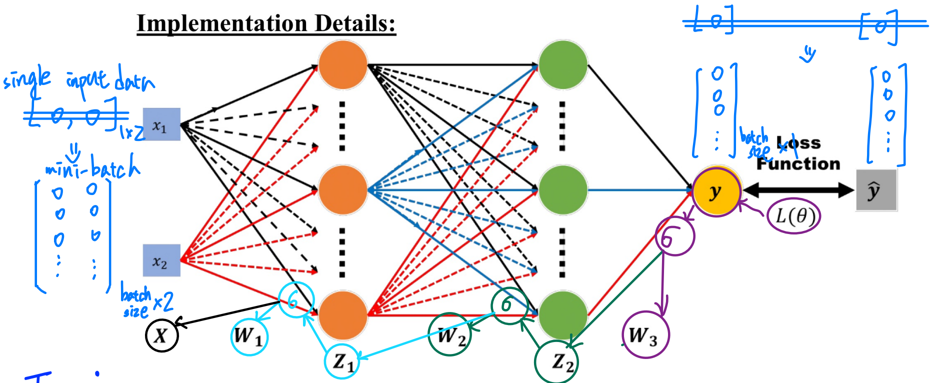



Figure 2. Forward pass

## 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 )

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

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

for each mini-batch

forward

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

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%):

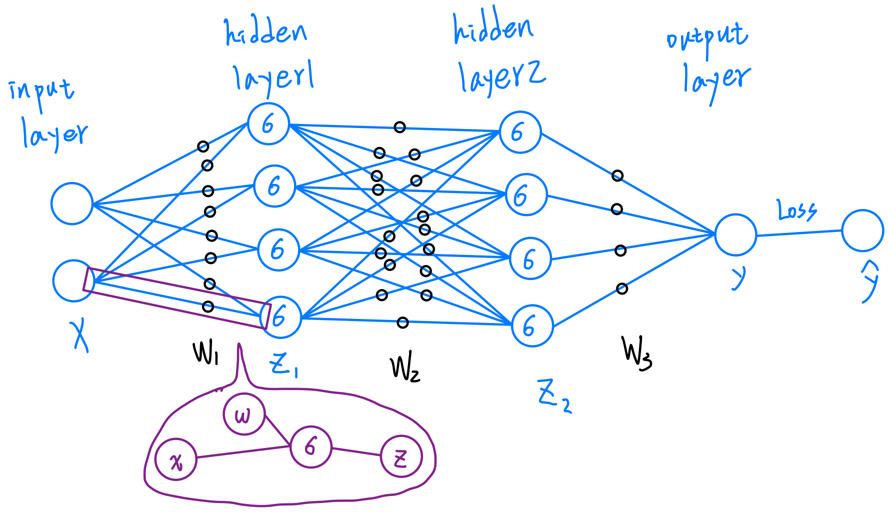
### A. Sigmoid functions

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

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

```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

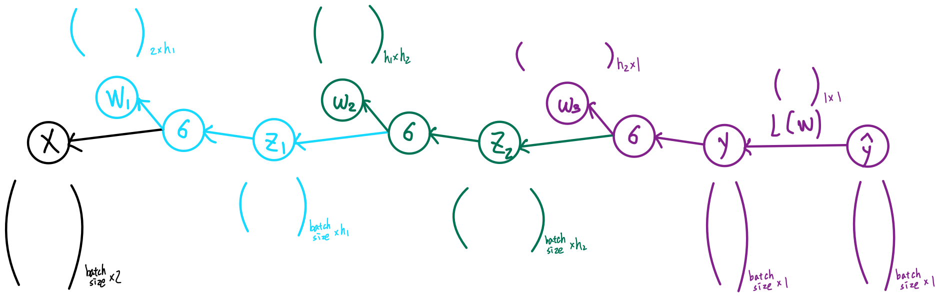
```python
def loss(self,gt_y,pred_y):
    batchsize = gt_y.shape[0]
    return (-1/batchsize) * np.sum(
        gt_y*np.log(gt_y+self.eps)
        +(1-pred_y)*np.log(1-pred_y+self.eps))
```

# C. Backpropogation

here shows the detailed backward computation using chain rule:

$\odot$: Hardamard product

@: standard matrix product




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

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

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

$$\sigma(Z_2W_3)_k(1-\sigma(Z_2W_3)_k) \qquad 0$$

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

has been computed

same technique (replace standard matrix product with Hardamard product)

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

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

same technique (replace standard matrix product with Hardamard product)

has been computed

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

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

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

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

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

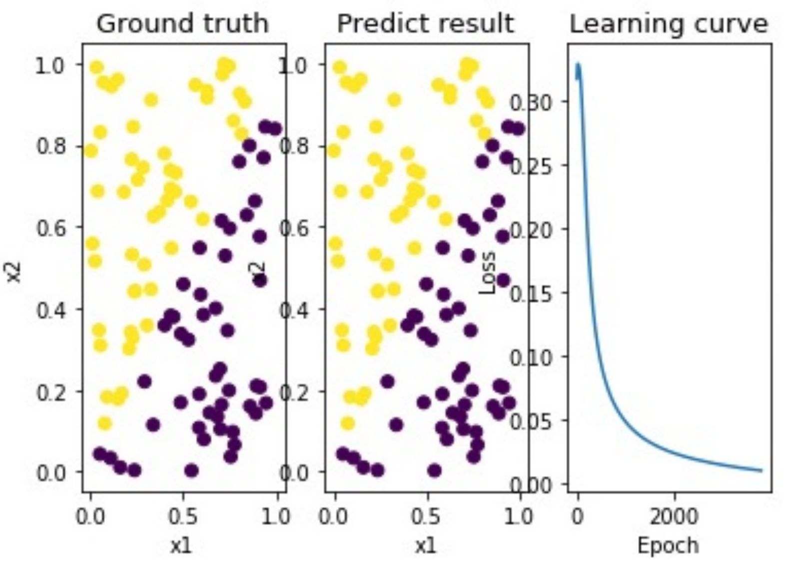
```python
def backward(self,gt_y,pred_y):
    #backward propogation
    #dL/d{W3} = dL/dy * dy/d{Z2W3} * d{Z2W3}/d{W3}
    batchsize = gt_y.shape[0]

    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%)

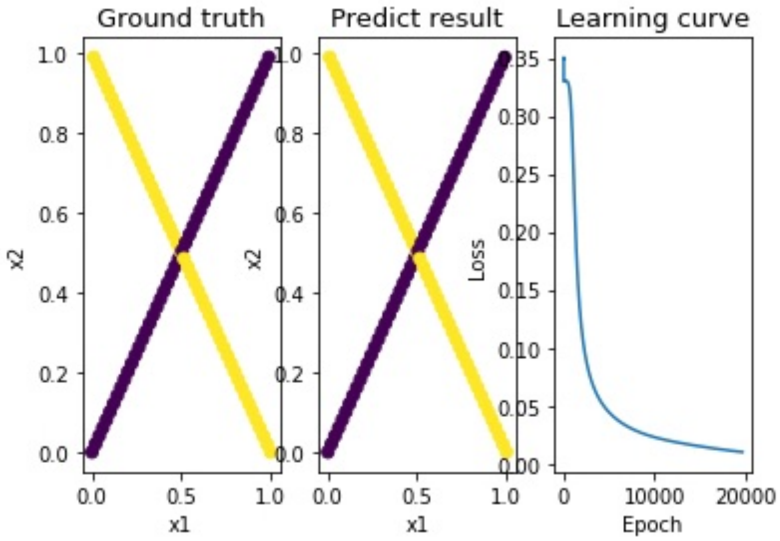
A. Screenshot and comparison figure

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
```

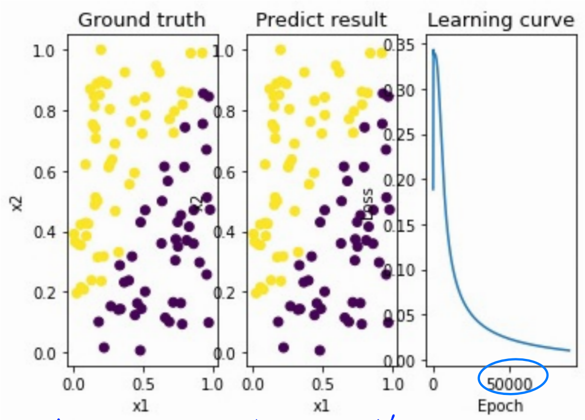
D. anything you want to present

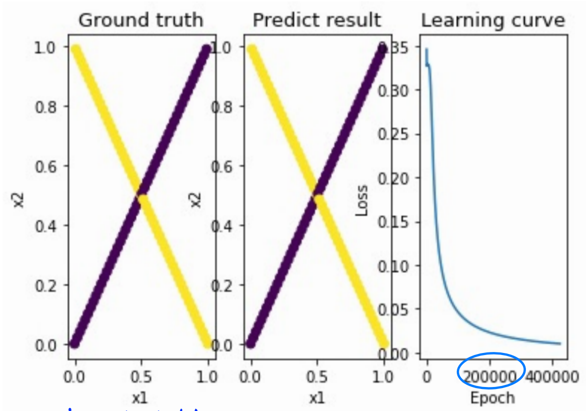
- in the backward propogation, because some matrices are actually diagonal matrices, the result of multiplying the diagonal matrices with other matrix is actually the same as directly multiplying two matrix's elements one by one. hence, in the implementation, I just use * instead of @.
- notice that 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%)

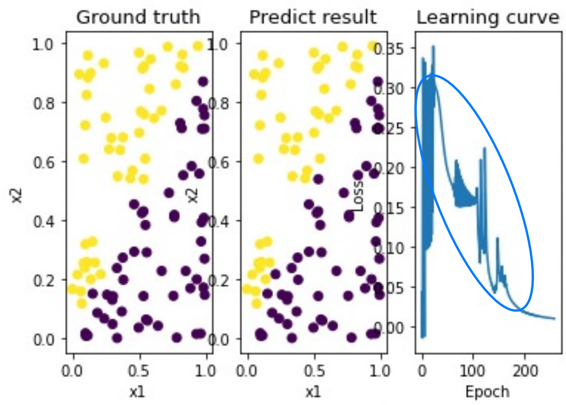
## A. Try different learning rates

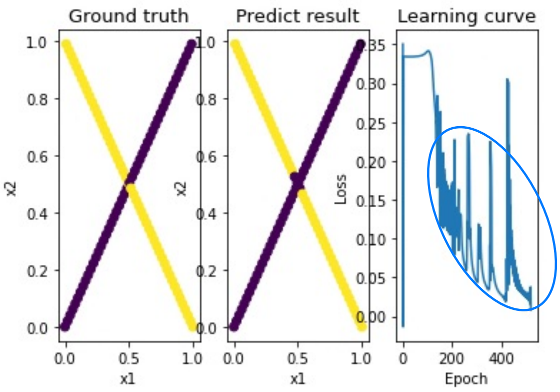
learning rate = 0.01, the network requires more epochs to converge



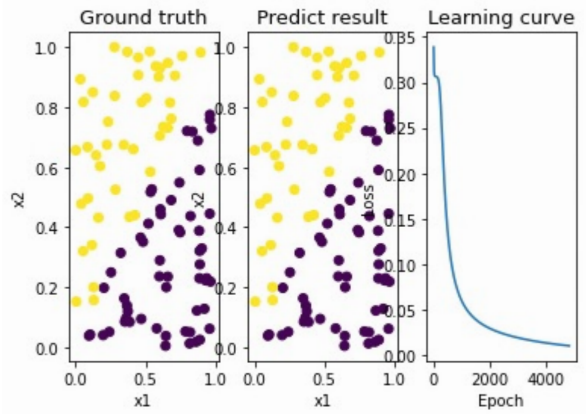

learning rate = 10, the network's convergence process is not stable

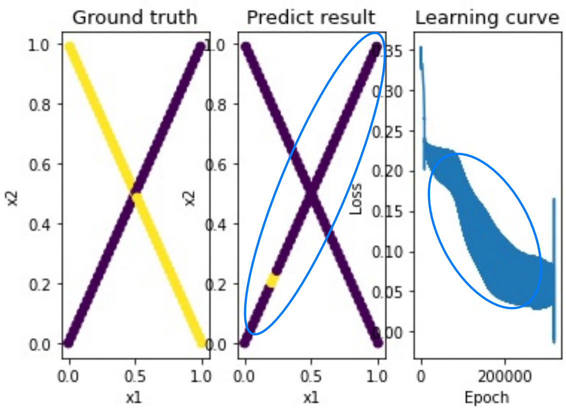


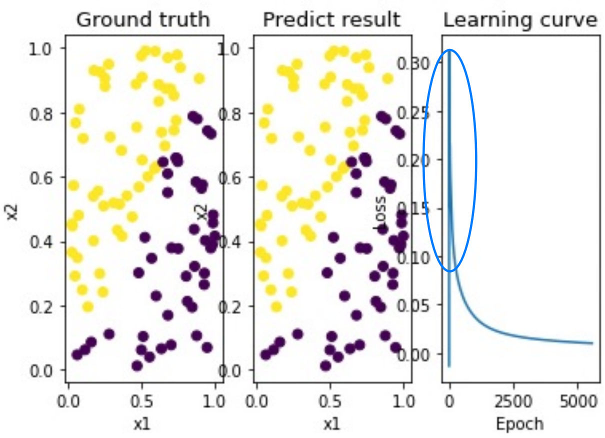

## B. Try different numbers of hidden units

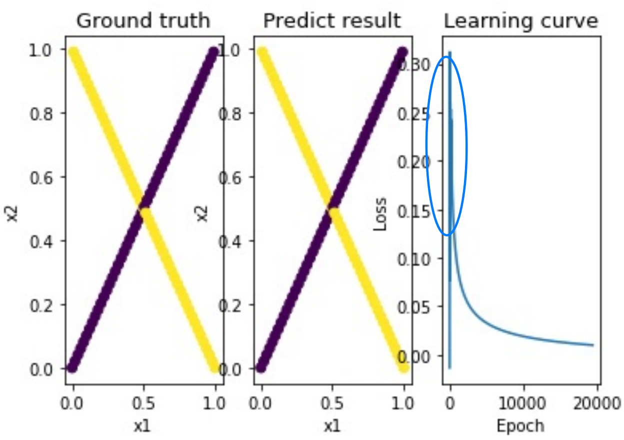
# of hidden units for first and second hidden layer = (3,3) , the network is not stable and may not predict well



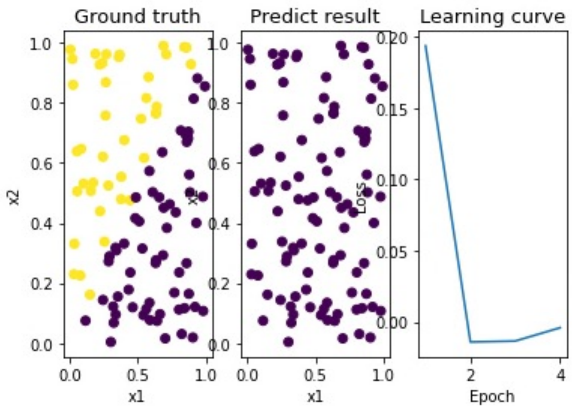



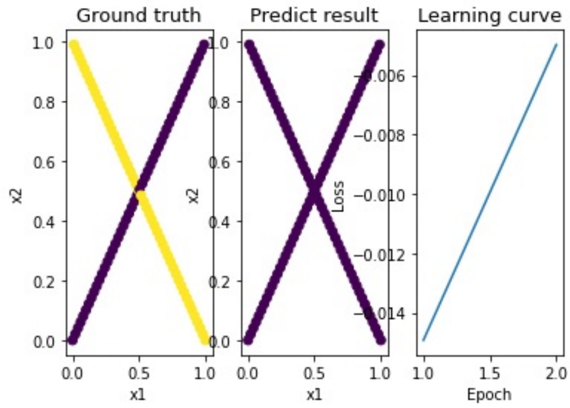

# of hidden units for first and second hidden layer = (70,70) , the loss decrease fast in the beginning



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




## C. Try without activation function

the 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
```