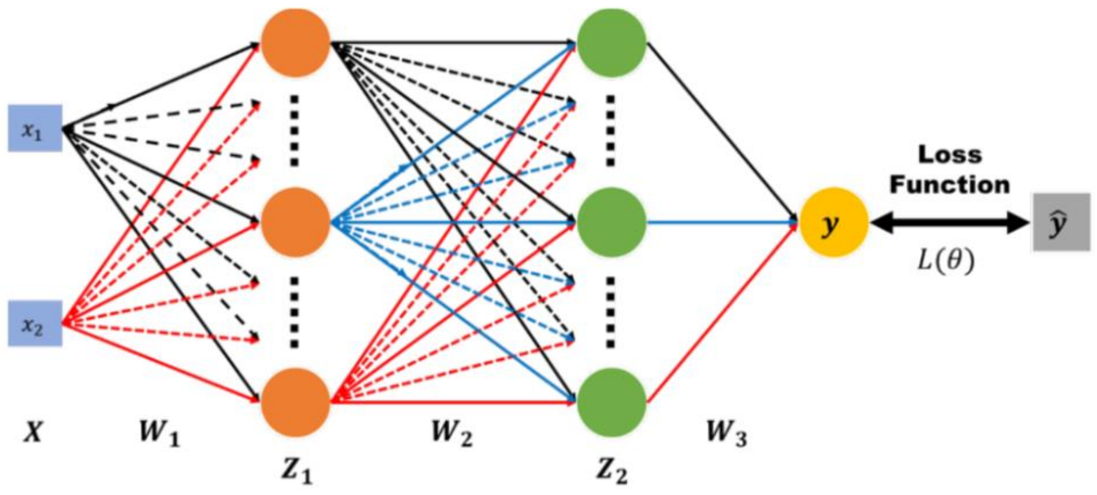
HW1 Report Backpropagation

tags: `DL and Practice`
電控碩一 黃柏叡 309512074

1 Introduction

建立一個有 2 層 hidden layers 的 neuron network 來 classify input data
loss function 使用 Mean-Square Error



1-1 Dataset

- Two Data Generator
    - Linear
    - XOR

```
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)
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)
def show_result(self, x, y, pred_y):
    count = 0
    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] < 0.5:
            plt.plot(x[i][0], x[i][1], "ro")
        else:
            plt.plot(x[i][0], x[i][1], "bo")

    for i in range(x.shape[0]):
        if(pred_y[i]-y[i]<0.1 and pred_y[i]-y[i]>-0.1):
            count+=1

    if(x.shape[0]==100):
        print("save linear")
        print("linear accuracy = ", count, "/", x.shape[0],"=",
float(count/x.shape[0]))
        plt.savefig("./images/linear.png")
        plt.close()

    else:
        print("save xor")
        print("xor accuracy = ", count, "/", x.shape[0],"=",
float(count/x.shape[0]))
        plt.savefig("./images/xor.png")
        plt.close()
```

2 Experiments setup

2-1 Sigmoid functions

I use Sigmoid functions as my activation function
Derivation of sigmoid is follow as:

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

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

$$= -(1+e^{-x})^2 \frac{d}{dx}(1+e^{-x})$$

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

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

```
def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))
def derivative_sigmoid(self,x):
    return np.multiply(x, 1-x)
```

2-2 Neural Network

neuron network with two hidden layers,
I use MSE(mean-suqre error) as my loss function

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

2-3 Backpropgation

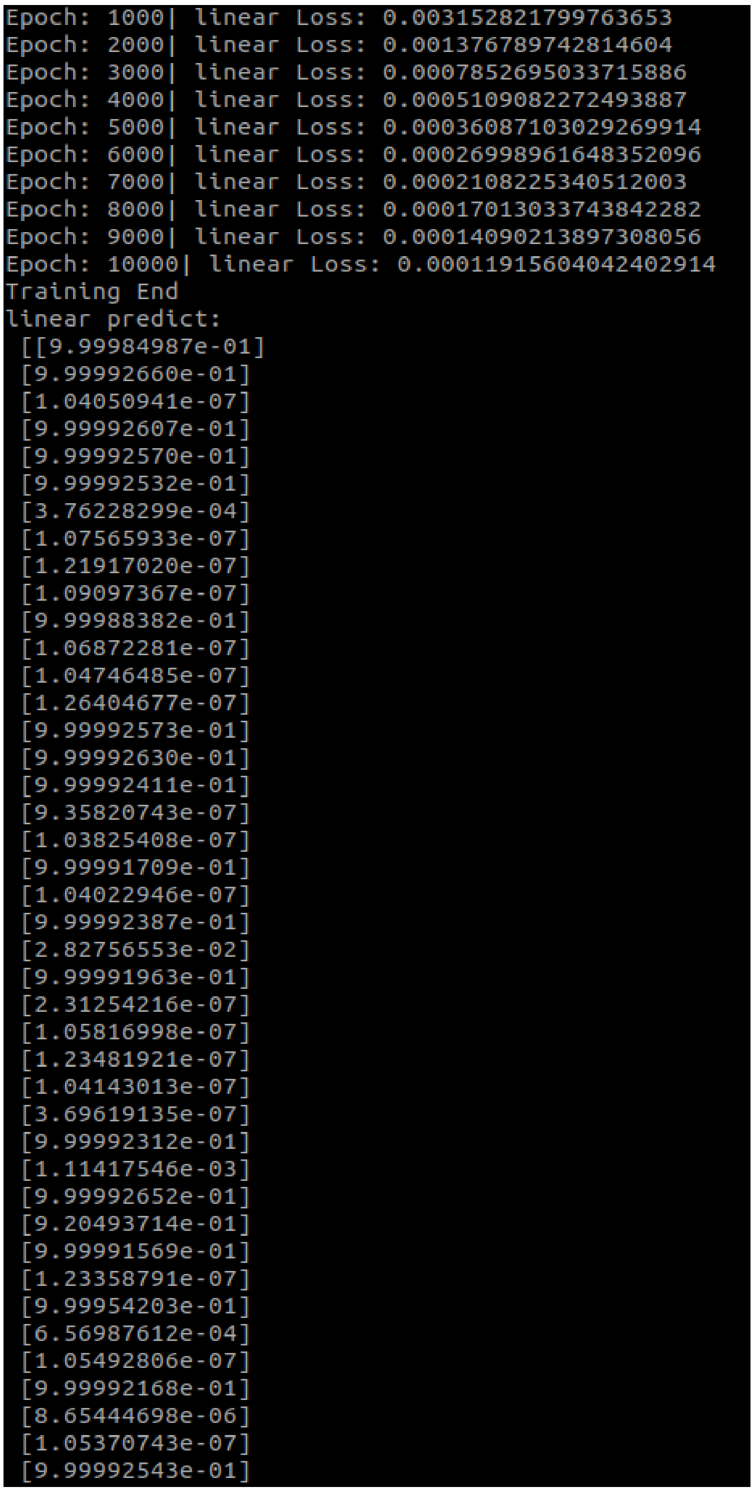
At first, all weight parameters are randomly initialized. The target is to minimize the cost from the loss function.
I use gradient descent to update the network’s weight, and calculate the gradient by chain rule.

```
def forward(self, x):
    self.x1 = np.dot(x, self.w1)
    self.a1 = self.sigmoid(self.x1)
    self.x2 = np.dot(self.a1, self.w2)
    self.a2 = self.sigmoid(self.x2)
    self.x3 = np.dot(self.a2, self.w3)
    self.y_pred = self.sigmoid(self.x3)
    return self.y_pred
    output = self.forward(x)
    loss = self.mseloss(output, y)
    self.loss_list.append(loss)

    #back propagation
    loss_grad = (output - y)
    w3_way = loss_grad *self.derivative_sigmoid(output)
    w3_grad = self.a2.T.dot(w3_way)
    w2_way = w3_way.dot(self.w3.T)*self.derivative_sigmoid(self.a2)
    w2_grad = self.a1.T.dot(w2_way)
    w1_way = w2_way.dot(self.w2.T)*self.derivative_sigmoid(self.a1)
    w1_grad = x.T.dot(w1_way)

    #update weights
    self.w1 = self.w1 - self.lr*w1_grad
    self.w2 = self.w2 - self.lr*w2_grad
    self.w3 = self.w3 - self.lr*w3_grad
```

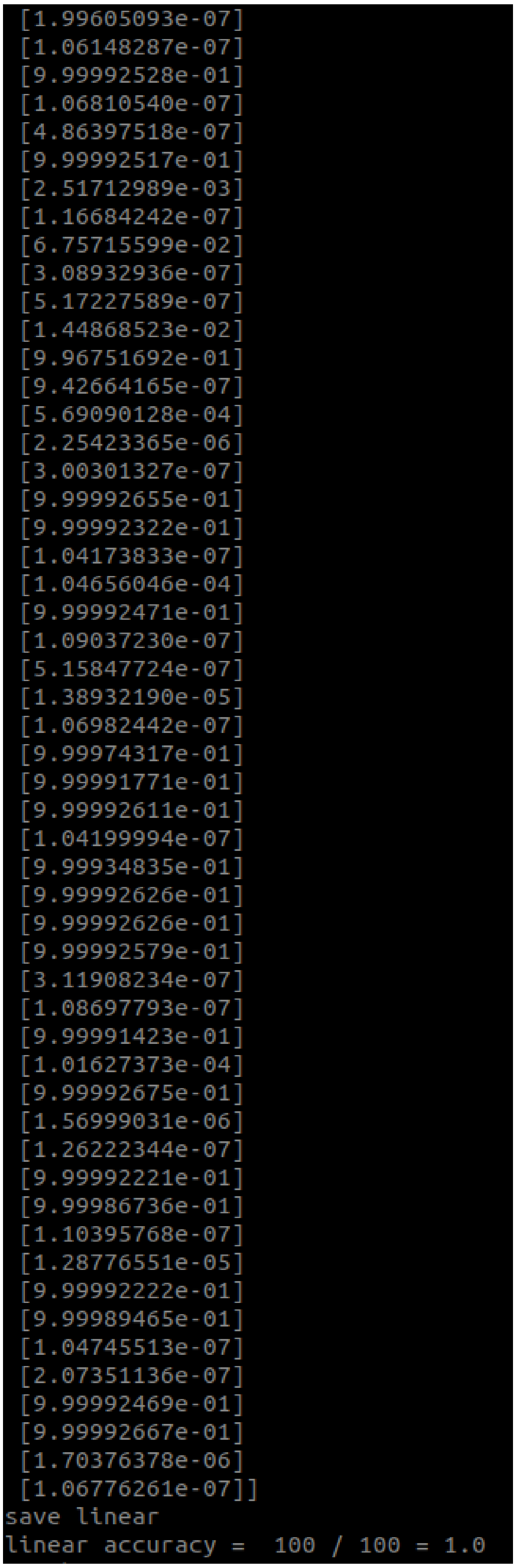
3 Results of your testing

3-1 Screenshot and comparison figure

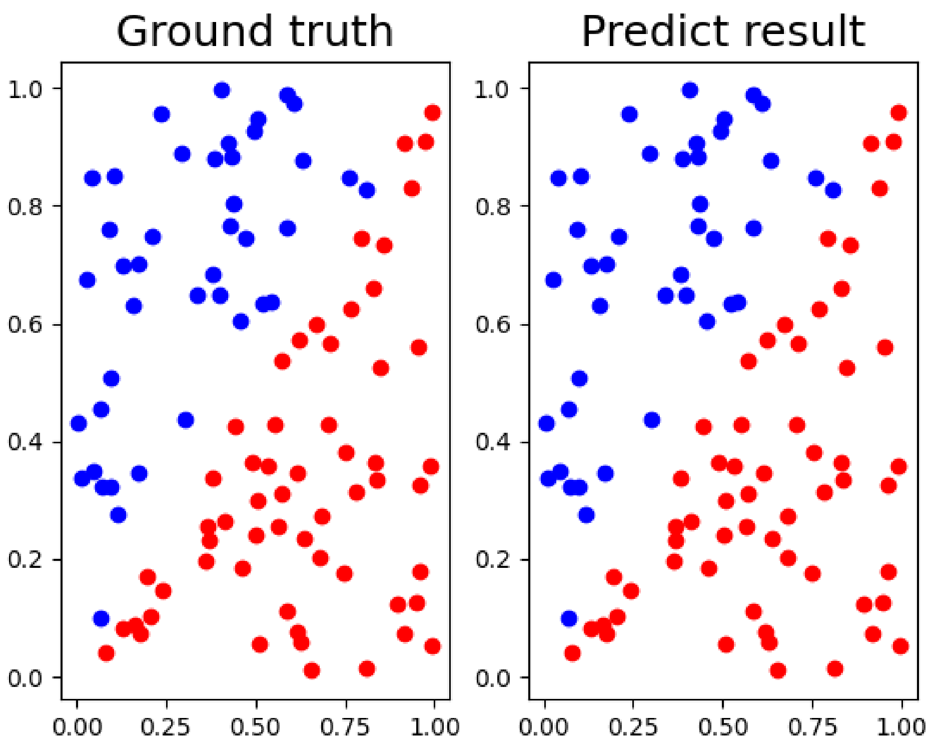
Linear:



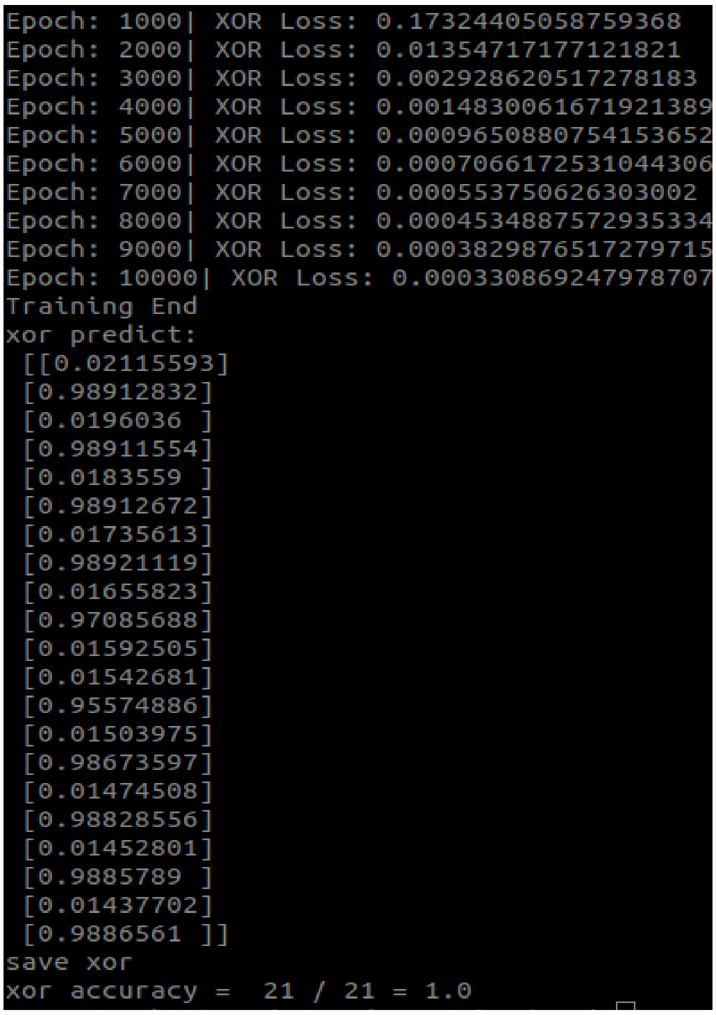
```
Epoch: 1000| linear Loss: 0.003152821799763653
Epoch: 2000| linear Loss: 0.001376789742814604
Epoch: 3000| linear Loss: 0.0007852695033715886
Epoch: 4000| linear Loss: 0.0005109082272493887
Epoch: 5000| linear Loss: 0.00036087103029269914
Epoch: 6000| linear Loss: 0.00026998961648352096
Epoch: 7000| linear Loss: 0.0002108225340512003
Epoch: 8000| linear Loss: 0.00017013033743842282
Epoch: 9000| linear Loss: 0.00014090213897308056
Epoch: 10000| linear Loss: 0.00011915604042402914
Training End
linear predict:
 [[9.99984987e-01]
 [9.99992660e-01]
 [1.04050941e-07]
 [9.99992607e-01]
 [9.99992570e-01]
 [9.99992532e-01]
 [3.76228299e-04]
 [1.07565933e-07]
 [1.21917020e-07]
 [1.09097367e-07]
 [9.99988382e-01]
 [1.06872281e-07]
 [1.04746485e-07]
 [1.26404677e-07]
 [9.99992573e-01]
 [9.99992630e-01]
 [9.99992411e-01]
 [9.35820743e-07]
 [1.03825408e-07]
 [9.99991709e-01]
 [1.04022946e-07]
 [9.99992387e-01]
 [2.82756553e-02]
 [9.99991963e-01]
 [2.31254216e-07]
 [1.05816998e-07]
 [1.23481921e-07]
 [1.04143013e-07]
 [3.69619135e-07]
 [9.99992312e-01]
 [1.11417546e-03]
 [9.99992652e-01]
 [9.20493714e-01]
 [9.99991569e-01]
 [1.23358791e-07]
 [9.99954203e-01]
 [6.56987612e-04]
 [1.05492806e-07]
 [9.99992168e-01]
 [8.65444698e-06]
 [1.05370743e-07]
 [9.99992543e-01]
```



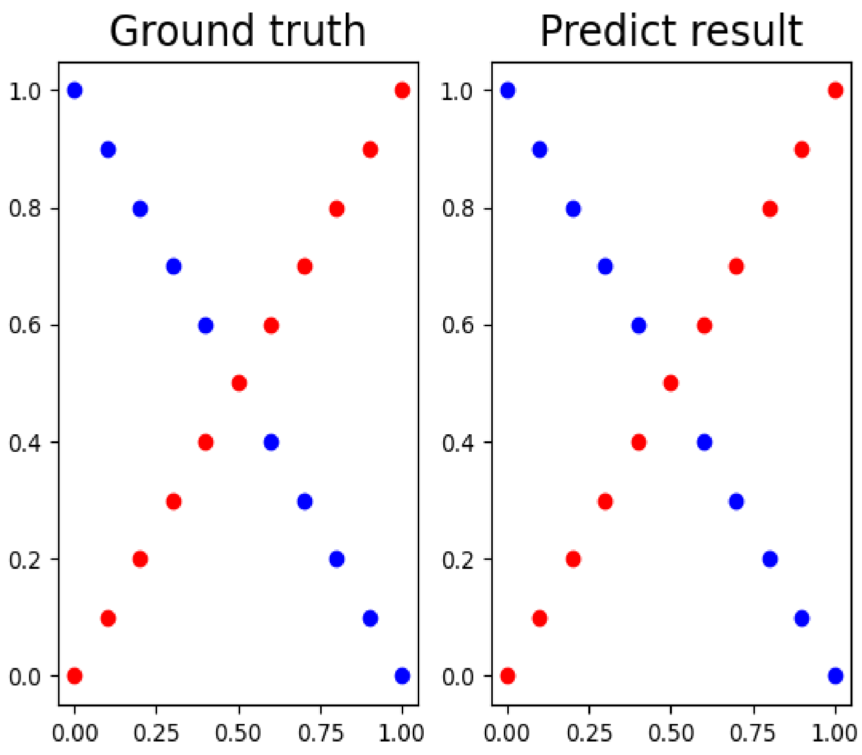
```
 [1.99605093e-07]
 [1.06148287e-07]
 [9.99992528e-01]
 [1.06810540e-07]
 [4.86397518e-07]
 [9.99992517e-01]
 [2.51712989e-03]
 [1.16684242e-07]
 [6.75715599e-02]
 [3.08932936e-07]
 [5.17227589e-07]
 [1.44868523e-02]
 [9.96751692e-01]
 [9.42664165e-07]
 [5.69090128e-04]
 [2.25423365e-06]
 [3.00301327e-07]
 [9.99992655e-01]
 [9.99992322e-01]
 [1.04173833e-07]
 [1.04656046e-04]
 [9.99992471e-01]
 [1.09037230e-07]
 [5.15847724e-07]
 [1.38932190e-05]
 [1.06982442e-07]
 [9.99974317e-01]
 [9.99991771e-01]
 [9.99992611e-01]
 [1.04199994e-07]
 [9.99934835e-01]
 [9.99992626e-01]
 [9.99992626e-01]
 [9.99992579e-01]
 [3.11908234e-07]
 [1.08697793e-07]
 [9.99991423e-01]
 [1.01627373e-04]
 [9.99992675e-01]
 [1.56999031e-06]
 [1.26222344e-07]
 [9.99992221e-01]
 [9.99986736e-01]
 [1.10395768e-07]
 [1.28776551e-05]
 [9.99992222e-01]
 [9.99989465e-01]
 [1.04745513e-07]
 [2.07351136e-07]
 [9.99992469e-01]
 [9.99992667e-01]
 [1.70376378e-06]
 [1.06776261e-07]]
save linear
linear accuracy =  100 / 100 = 1.0
```


Ground truth
Predict result
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00

XOR:



```
Epoch: 1000| XOR Loss: 0.17324405058759368
Epoch: 2000| XOR Loss: 0.01354717177121821
Epoch: 3000| XOR Loss: 0.002928620517278183
Epoch: 4000| XOR Loss: 0.0014830061671921389
Epoch: 5000| XOR Loss: 0.0009650880754153652
Epoch: 6000| XOR Loss: 0.0007066172531044306
Epoch: 7000| XOR Loss: 0.000553750626303002
Epoch: 8000| XOR Loss: 0.0004534887572935334
Epoch: 9000| XOR Loss: 0.0003829876517279715
Epoch: 10000| XOR Loss: 0.0003308692479787O7
Training End
xor predict:
 [[0.02115593]
 [0.98912832]
 [0.0196036 ]
 [0.98911554]
 [0.0183559 ]
 [0.98912672]
 [0.01735613]
 [0.98921119]
 [0.01655823]
 [0.97085688]
 [0.01592505]
 [0.01542681]
 [0.95574886]
 [0.01503975]
 [0.98673597]
 [0.01474508]
 [0.98828556]
 [0.01452801]
 [0.9885789 ]
 [0.01437702]
 [0.9886561 ]]
save xor
xor accuracy =  21 / 21 = 1.0
```

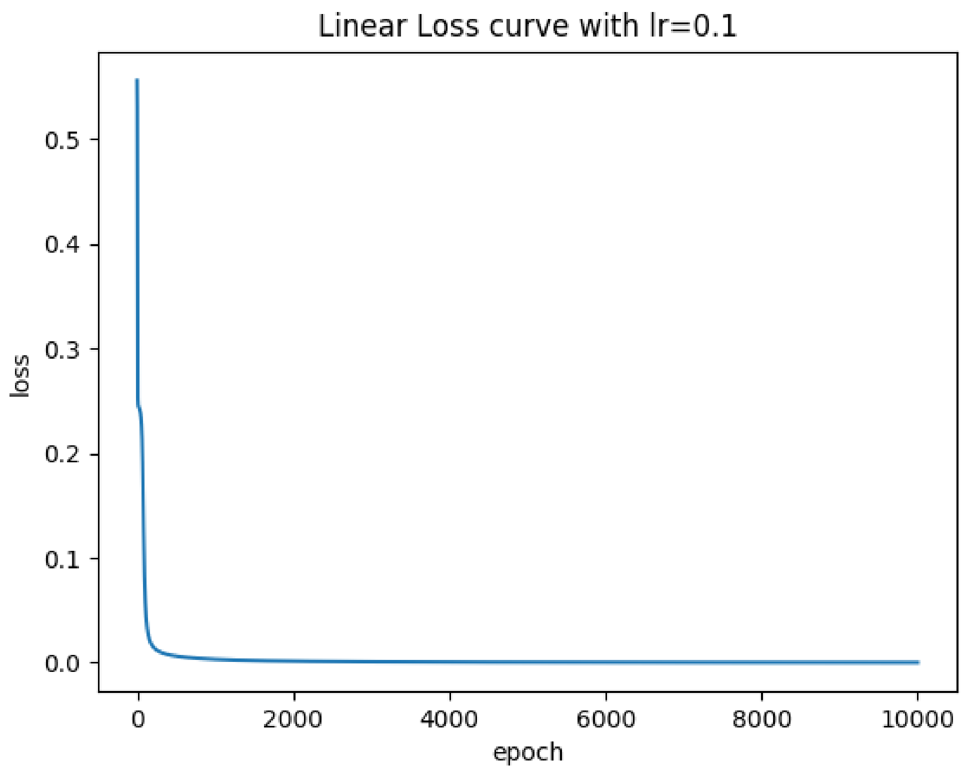


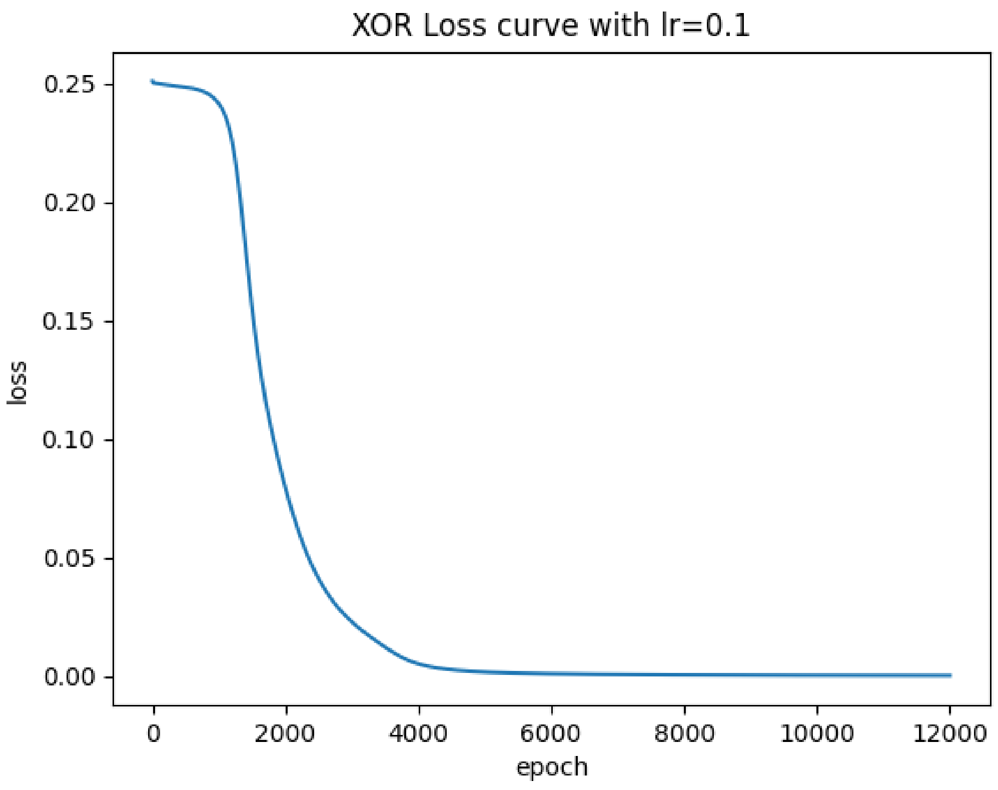
3-2 Show the accuracy of your prediction

y[i] : target output, pred_y[i] : predict output, the accuracy was calculated when the error is smaller than 0.1
both accuracy of linear and XOR is 100% as shown in the figure above.
Besides, the ground truth and predict result figure show that the accuracy is 100%

```
for i in range(x.shape[0]):
    if(pred_y[i]-y[i]<0.1 and pred_y[i]-y[i]>-0.1):
        count+=1
```

3-3 Learning curve(loss, epoch curve)





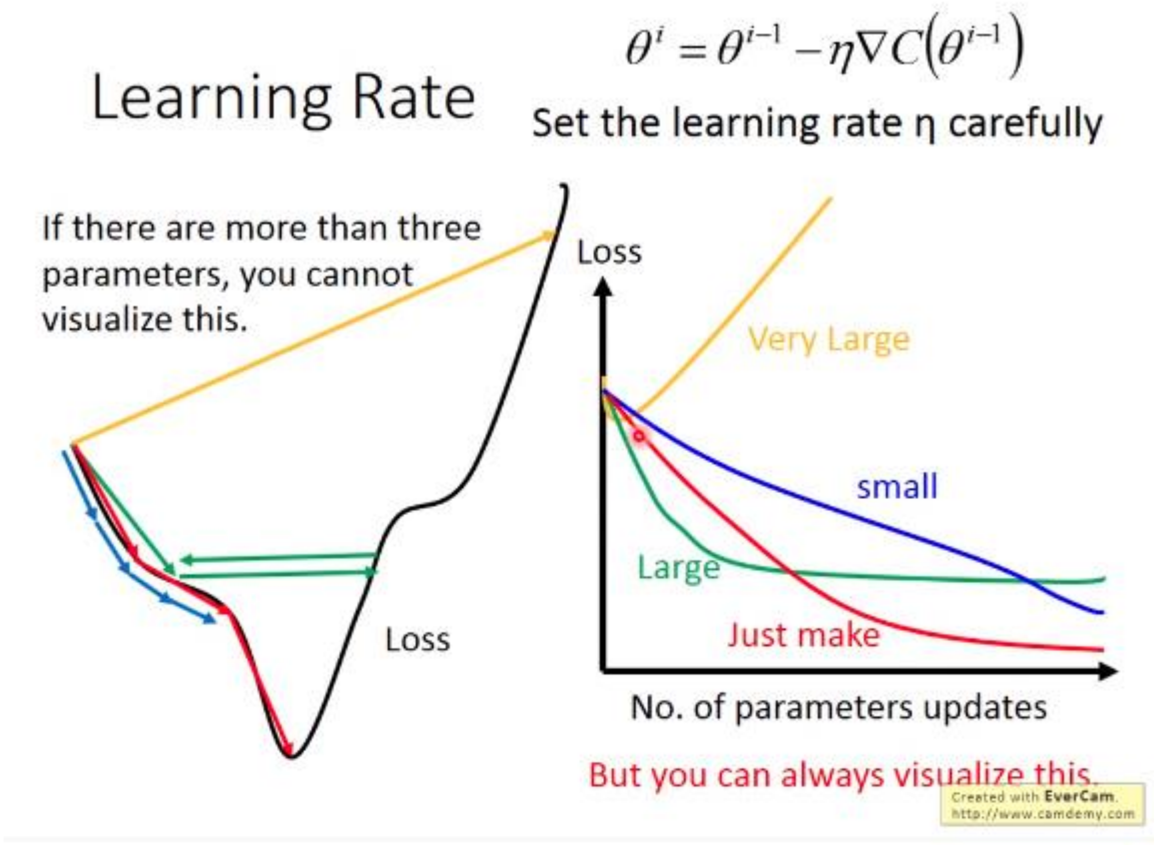
3-4 anything you want to share

All weights are independent, at first I didn’t notice this detail, therefore the backpropagation I derived was wrong, later I slove the problem and derived the correct backpropagation by chain rule.

4 Disscussion

4-1 Try different learning rates

I need to choose a suitable learning rate for training in order to reach the minimum in a acceptable time


Learning Rate
$\theta^i = \theta^{i-1} - \eta \nabla C(\theta^{i-1})$
Set the learning rate η carefully
If there are more than three parameters, you cannot visualize this.
Loss
Loss
Very Large
small
Large
Just make
No. of parameters updates
But you can always visualize this.
Created with EverCam. http://www.camdemy.com

learning rate = 0.1, linear_epoch = 10000, xor_epoch = 12000 to achieve 100% accuracy

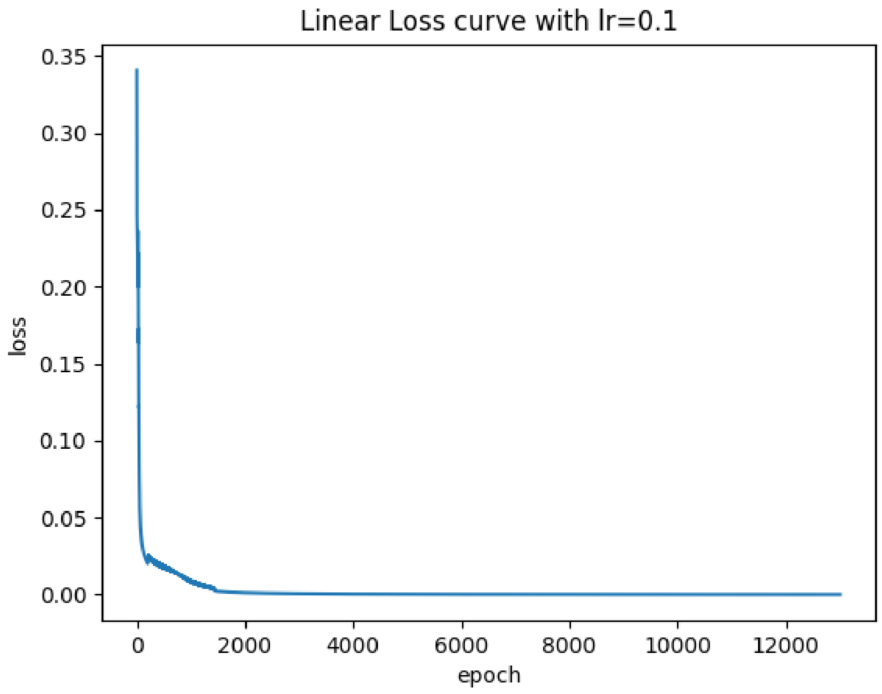
learning rate = 0.05, linear_epoch = 22000, xor_epoch = 20000 to achieve 100% accuracy

4-2 Try different number of hidden units

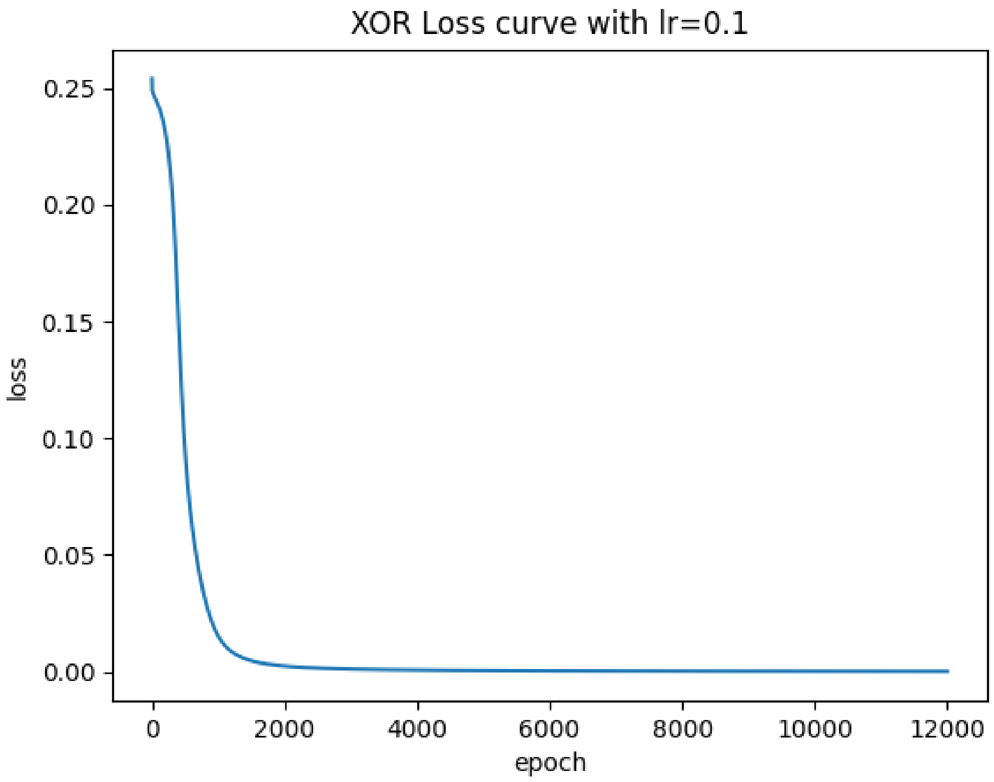
For the pictures shown above, the hidden layers I chose was hid1=4 hid2=4

(2) I pick hid1=10 hid2=10, and change linear_epoch=13000 to achieve 100% accuracy

Linear


Linear Loss curve with lr=0.1
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
loss
0
2000
4000
6000
8000
10000
12000
epoch

XOR


XOR Loss curve with lr=0.1
0.25
0.20
0.15
0.10
0.05
0.00
loss
0
2000
4000
6000
8000
10000
12000
epoch

4-3 Try without activation functions

```
def forward(self, x):
    self.x1 = np.dot(x, self.w1)
    self.x2 = np.dot(self.x1, self.w2)
    self.x3 = np.dot(self.x2, self.w3)
    self.y_pred = self.x3
    return self.y_pred
```

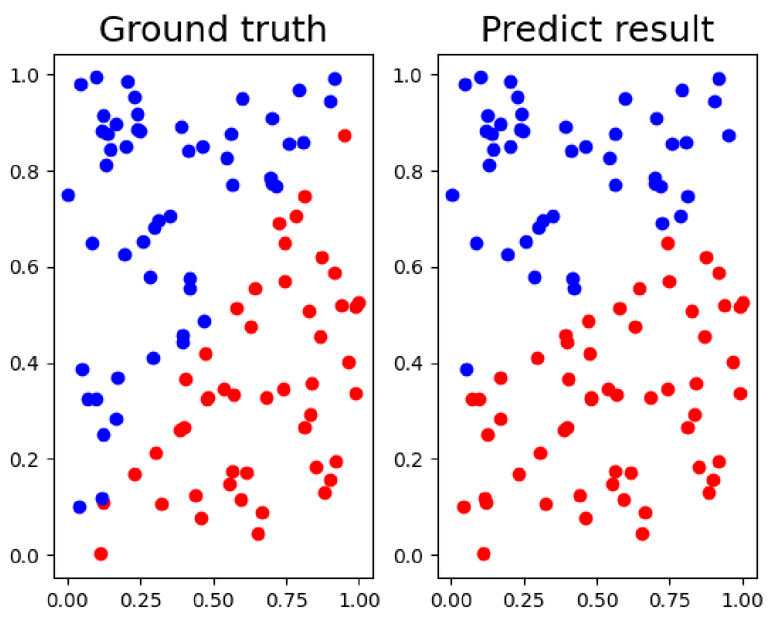
back propagation without activation

```
    loss_grad = (output - y)
    w3_way = loss_grad
    w3_grad = self.x2.T.dot(w3_way)
    w2_way = w3_way.dot(self.w3.T)
    w2_grad = self.x1.T.dot(w2_way)
    w1_way = w2_way.dot(self.w2.T)
    w1_grad = x.T.dot(w1_way)
```

First I change the learning rate to 0.001 otherwise overshoot problem will occur.

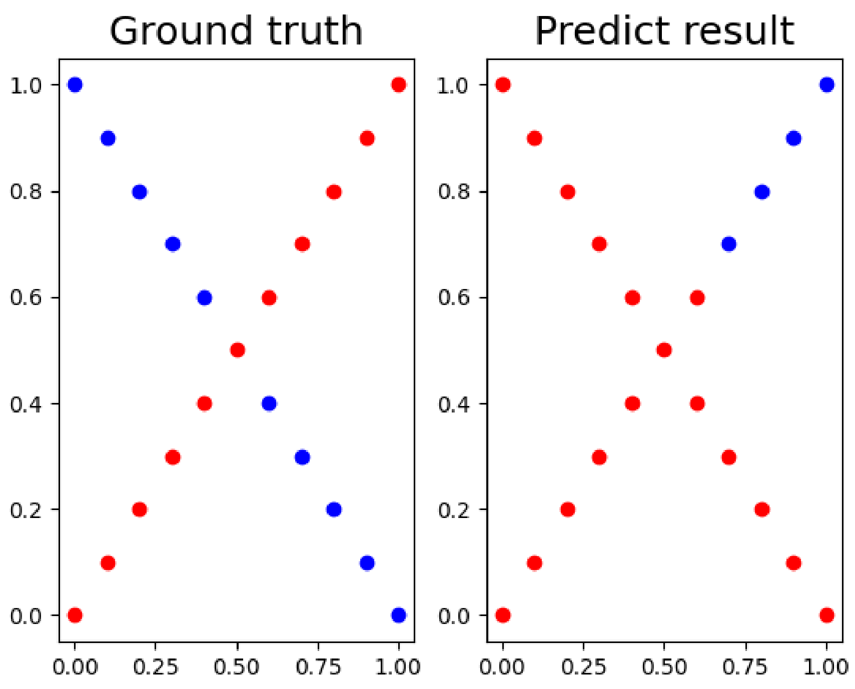
linear without activation funciton

For linear problem, we can roughly differentiate the data into two classes, therefore the predict result is roughly correct



xor without activation function

However, for xor problem, we can’t differentiate the data by linear function, therefore the predict result has huge error. To conclude, since most of the data are non-linear, activation functions are very important in training.



4-4 anything you want to share

5 Extra

5-1 Implement different optimizers

5-2 Implement different activation functions

```
def relu(self, x):
    return np.maximum(0, x)

def derivative_relu(self, x):
    z = np.copy(x)
    z[z>0]=1
    z[z<=0]=0
    return z
```
However, my training failed if I use relu as my activation function

5-3 Implement convolutional layers