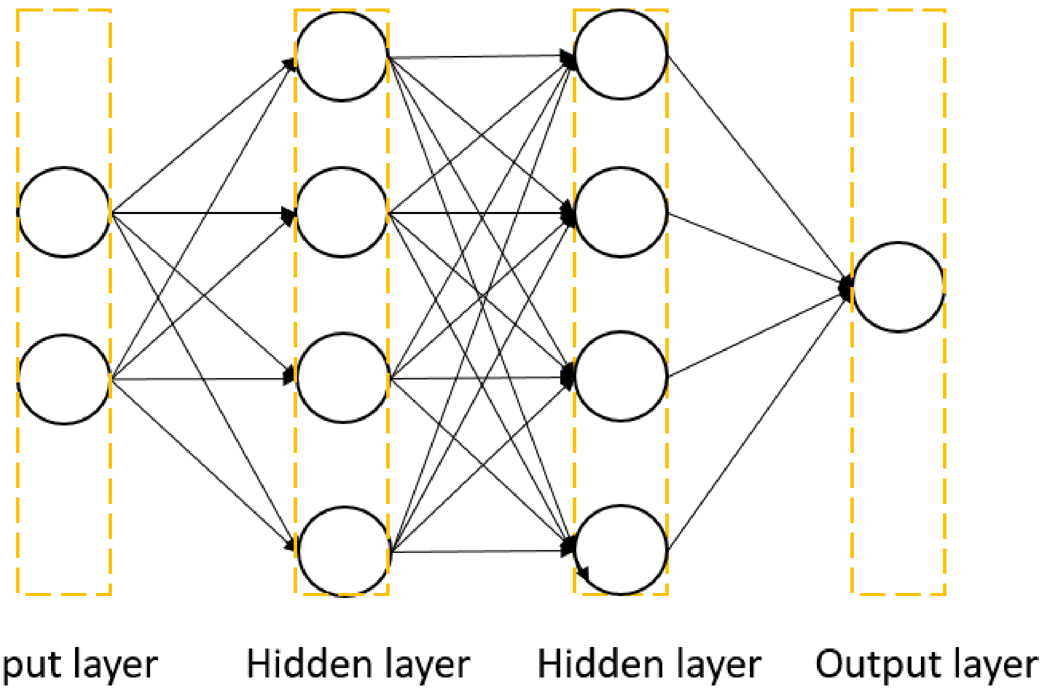
Lab1 : back-propagation

1. Introduction

Lab Objective:

In this lab, you will need to understand and write a simple neural networks with forward pass and back propagation using two hidden layers. Noticed that you only can use Numpy and other python standard library, any other framework (ex : Tensorflow, PyTorch) is not allowed in this lab.



So basically this homework is asking as to implement a neural network from scratch by our self without any third parties' library. The simple neural network function has **three** main parts:

1. Initialization
2. Forward pass
3. Back propagation.

-Initialization part helps to create a model with specific layers and initialize the weight of layers.

-Forward pass part pass the input data through the model and get the predict output.

-Back propagation part calculates the gradients by the difference between predict output and the expected output. Then update the weight of layers by these gradients.
More details and the implementation code will be shown in the below sections.

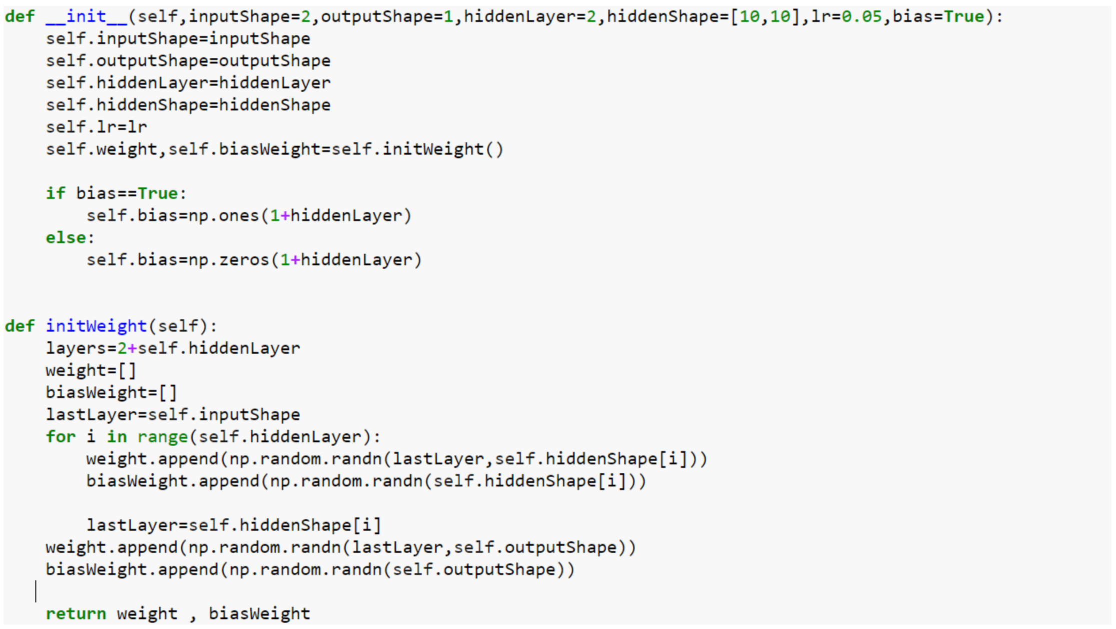
2. Experiment setups

A. Sigmoid functions

This function has already provided by the spec. I just use it directly without change any thing.

B. Neural network

This part contain two components: Initialization and Forward pass:
Initialization:



```python
def __init__(self,inputShape=2,outputShape=1,hiddenLayer=2,hiddenShape=[10,10],lr=0.05,bias=True):
    self.inputShape=inputShape
    self.outputShape=outputShape
    self.hiddenLayer=hiddenLayer
    self.hiddenShape=hiddenShape
    self.lr=lr
    self.weight,self.biasWeight=self.initWeight()

    if bias==True:
        self.bias=np.ones(1+hiddenLayer)
    else:
        self.bias=np.zeros(1+hiddenLayer)


def initWeight(self):
    layers=2+self.hiddenLayer
    weight=[]
    biasWeight=[]
    lastLayer=self.inputShape
    for i in range(self.hiddenLayer):
        weight.append(np.random.randn(lastLayer,self.hiddenShape[i]))
        biasWeight.append(np.random.randn(self.hiddenShape[i]))

        lastLayer=self.hiddenShape[i]
    weight.append(np.random.randn(lastLayer,self.outputShape))
    biasWeight.append(np.random.randn(self.outputShape))

    return weight , biasWeight
```

Create a simple neural network model with specific hyper parameters, the default values for input size, output size and the number of layers are follow the spec of the homework, other default values are just set randomly.

The weights of each layers are define as follow: if the previous layer's output dimension is 2 and the next layer's input dimension is 4, the weights between these layers are represent will a matrix with shape 2*4, something like this:

$$\begin{bmatrix} Wa1b1 & Wa1b2 & Wa1b3 & Wa1b4 \\ Ww21 & Wa2b2 & Wa2b3 & Wa2b4 \end{bmatrix}$$

a and b is the neural of the previous layer and next layer. Waxby indicates the weight between ax neural and by neural.

The initial values of the weight are set by using the randn() function from **Numpy**, which will return a sample (or samples) from the "standard normal" distribution.

Forward pass:

```
def forward(self,x):
    self.y=[]
    _x=x
    self.y.append(_x)
    for i in range(len(self.weight)):
        if(_x.shape[-1]!=self.weight[i].shape[0]):
            print("In layer %d , dim %d and dim %d does not match" %(i,_x.shape[-1],self.weight[i].shape[0]))
        _x=sigmoid(_x@self.weight[i]+self.bias[i]*self.biasWeight[i])
        self.y.append(_x)
    return np.array(self.y)
```

Nothing really special here, just a simple matrix multiplication. I choose Sigmoid as the activation function after each layers. This function pass the input (x) to the model and get the predict output.

C. Back propagation

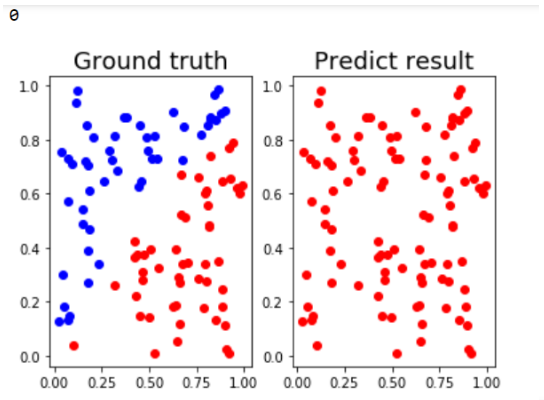
```
def backward(self,loss):
    delta=(loss)*derivative_sigmoid(self.y[-1])
    for i in range(1,len(self.weight)+1):
        tmp=delta.copy()
        delta=(delta@self.weight[-i].transpose())*derivative_sigmoid(self.y[-i-1])
        self.weight[-i]=self.weight[-i]-self.lr*(self.y[-i-1].reshape(-1,1)@tmp.reshape(1,-1))
        self.biasWeight[-i]=self.biasWeight[-i]-self.lr*tmp
```

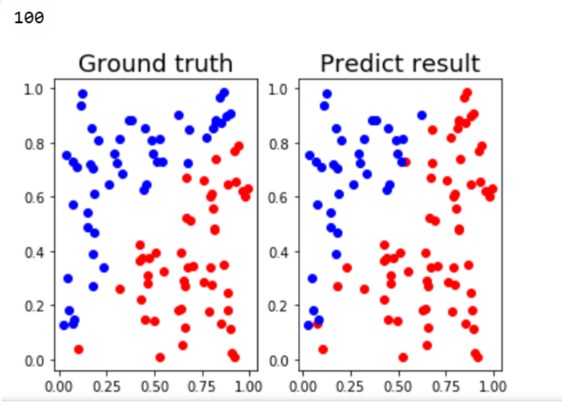
Spec has already given a perfect detail of this part, also there is a pseudo code. So I am not going to make any explanation here.

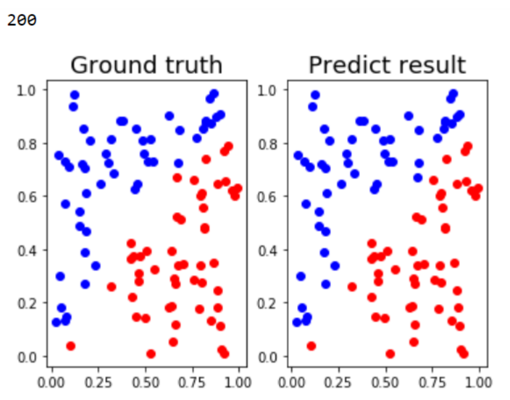
3. Results of your testing

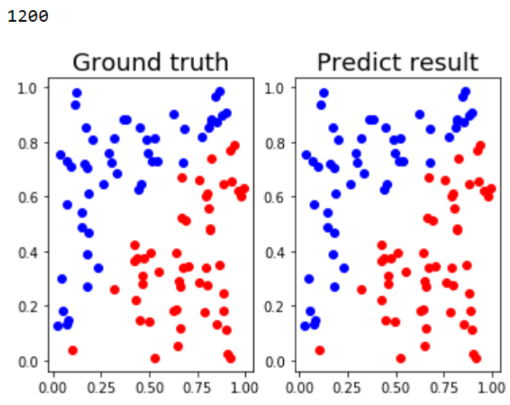
A. Comparison figure

Linear

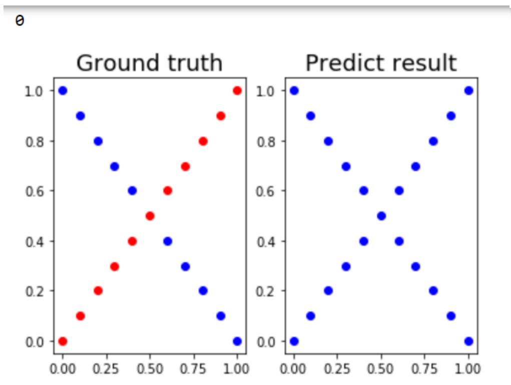

```
0
```

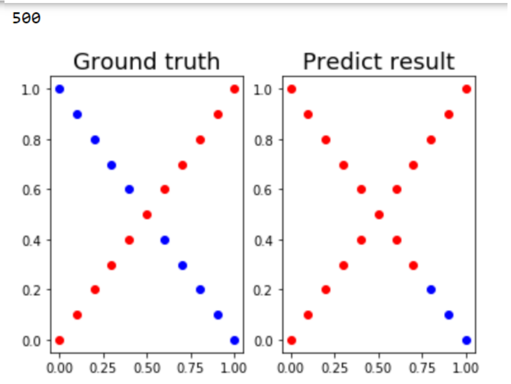

```
100
```

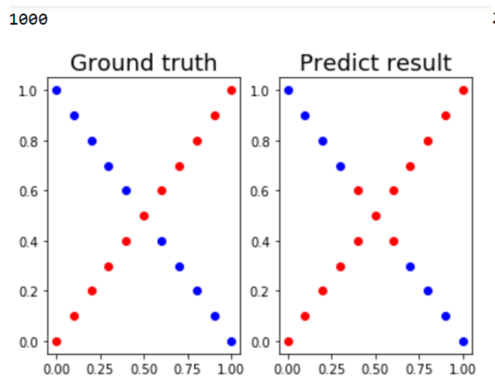

```
200
```

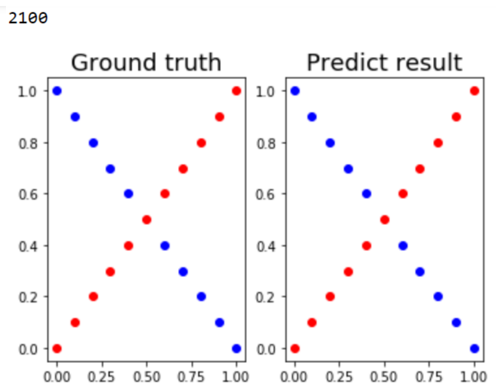

```
1200
```

XOR

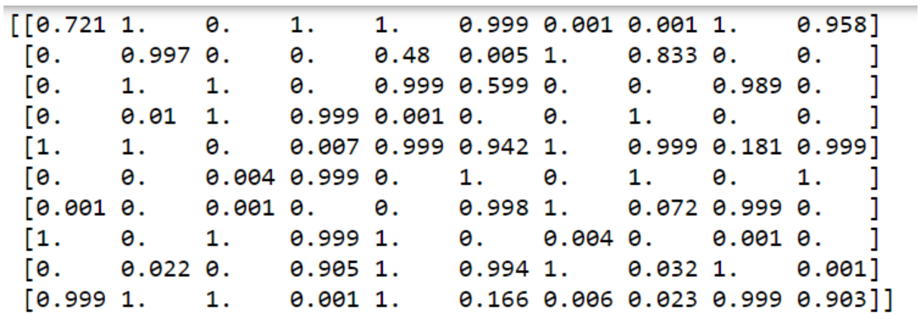

```
0
```


```
500
```

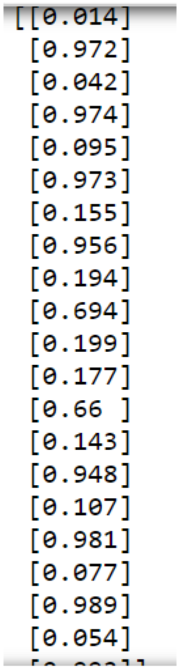




B. Predictions of model(round to 3 digits)

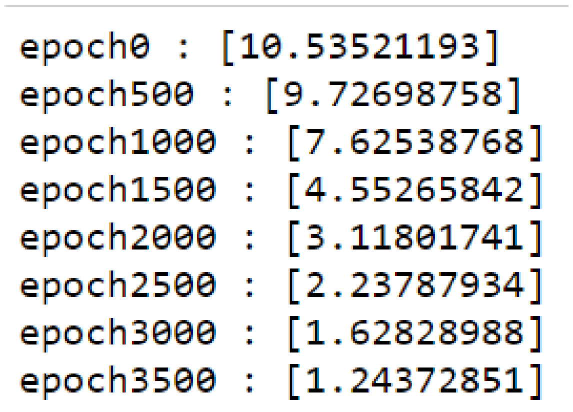


```
[[0.721 1.    0.    1.    1.    0.999 0.001 0.001 1.    0.958]
 [0.    0.997 0.    0.    0.48  0.005 1.    0.833 0.    0.   ]
 [0.    1.    1.    0.    0.999 0.599 0.    0.    0.989 0.   ]
 [0.    0.01  1.    0.999 0.001 0.    0.    1.    0.    0.   ]
 [1.    1.    0.    0.007 0.999 0.942 1.    0.999 0.181 0.999]
 [0.    0.    0.004 0.999 0.    1.    0.    1.    0.    1.   ]
 [0.001 0.    0.001 0.    0.    0.998 1.    0.072 0.999 0.   ]
 [1.    0.    1.    0.999 1.    0.    0.004 0.    0.001 0.   ]
 [0.    0.022 0.    0.905 1.    0.994 1.    0.032 1.    0.001]
 [0.999 1.    1.    0.001 1.    0.166 0.006 0.023 0.999 0.903]]
```

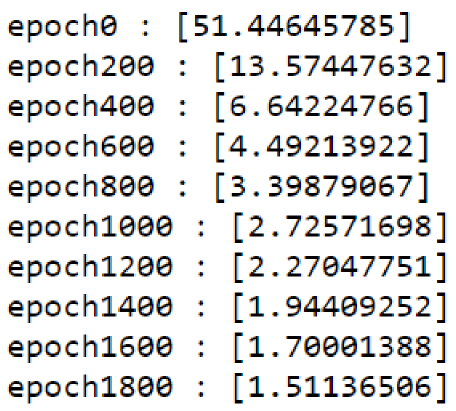


```
[[0.014]
 [0.972]
 [0.042]
 [0.974]
 [0.095]
 [0.973]
 [0.155]
 [0.956]
 [0.194]
 [0.694]
 [0.199]
 [0.177]
 [0.66 ]
 [0.143]
 [0.948]
 [0.107]
 [0.981]
 [0.077]
 [0.989]
 [0.054]
```

C. Loss



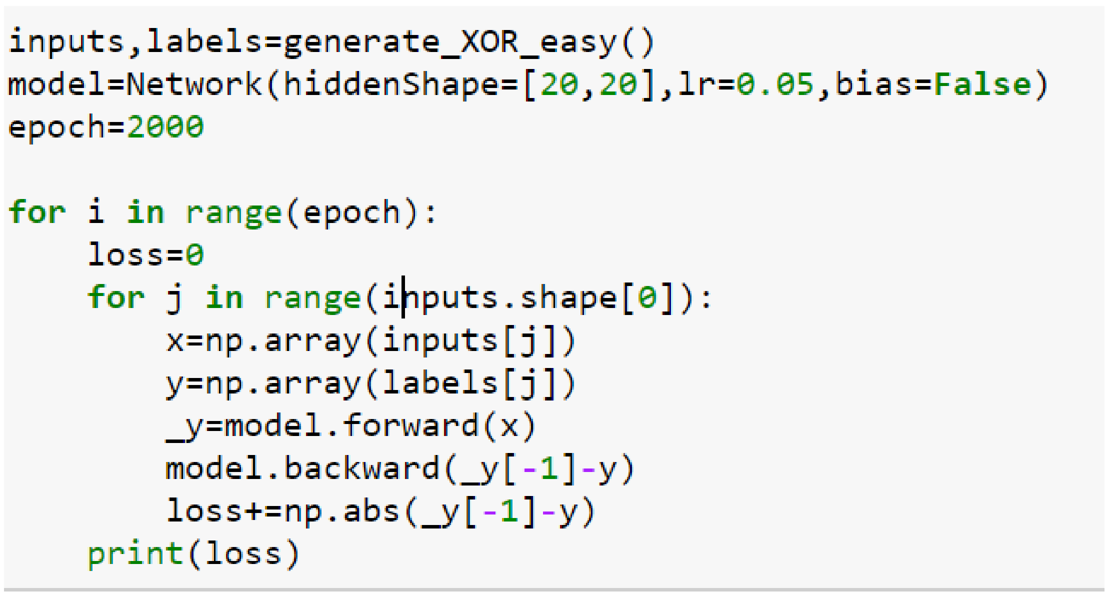
```
epoch0 : [10.53521193]
epoch500 : [9.72698758]
epoch1000 : [7.62538768]
epoch1500 : [4.55265842]
epoch2000 : [3.11801741]
epoch2500 : [2.23787934]
epoch3000 : [1.62828988]
epoch3500 : [1.24372851]
```



```
epoch0 : [51.44645785]
epoch200 : [13.57447632]
epoch400 : [6.64224766]
epoch600 : [4.49213922]
epoch800 : [3.39879067]
epoch1000 : [2.72571698]
epoch1200 : [2.27047751]
epoch1400 : [1.94409252]
epoch1600 : [1.70001388]
epoch1800 : [1.51136506]
```

B. anything you want to present

The code of training part:



```
inputs,labels=generate_XOR_easy()
model=Network(hiddenShape=[20,20],lr=0.05,bias=False)
epoch=2000

for i in range(epoch):
    loss=0
    for j in range(inputs.shape[0]):
        x=np.array(inputs[j])
        y=np.array(labels[j])
        _y=model.forward(x)
        model.backward(_y[-1]-y)
        loss+=np.abs(_y[-1]-y)
    print(loss)
```

I use the exactly same hyper parameters for both input. Also because I am quite lazy, I didn't use any strategy(such as grid such) to find the hyper parameters , I just randomly decided some value and ran my code. The result turned out to be quite good. So I stick with this set of the hyper parameters, I think there is a better choice to reach the same result with less epochs.

The training loss and the predict values for can be seen in my attachment (IPython)

4. Discussion

A. Anything you want to share

Nothing I really want to share, this homework is quite easy (if I did not misunderstand what the purpose of this homework). Also because the training data and the testing data are the same in this homework, we can just randomly pick a hyper parameters and it will work I think.

the full code can be seen in my attachment or on my github:

https://github.com/s410385015/Deep-Learning-HW