NCTU Deep Learning Lab1 Report

309833023 簡廷羽

I. Introduction

This lab aims at building a neural network with two hidden layers. Our mainly mission is to implement above parameters and two function ”forward” and “backward”.

Also, we need to print the plot of our experimental results and try to compare with its ground truth.

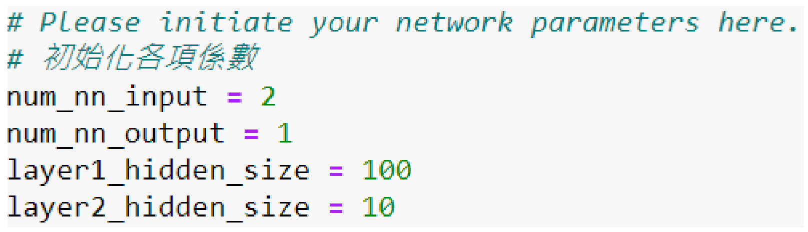
II. Experimental Setup

1. Sigmoid function：

   We’re using sigmoid function to scale our output to range [0,1] by formula $\frac{1}{1+e^{-x}}$, there’s no need to modify the giving code.

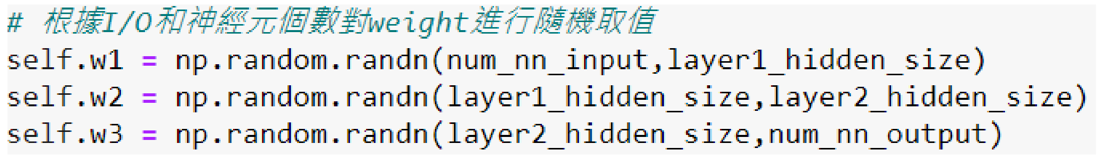
2. Neural network：

   This network is fed by two parameters: steps and batch size, and we’re focus on initiating condition also by parameters



```
# Please initiate your network parameters here.
# 初始化各項係數
num_nn_input = 2
num_nn_output = 1
layer1_hidden_size = 100
layer2_hidden_size = 10
```

   and giving appropriate weights to the network.

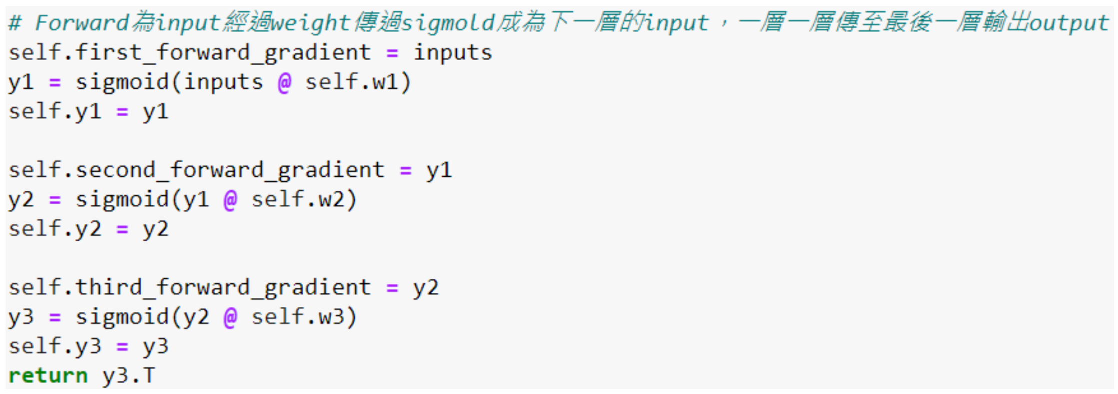


```
# 根據I/O和神經元個數對weight進行隨機取值
self.w1 = np.random.randn(num_nn_input,layer1_hidden_size)
self.w2 = np.random.randn(layer1_hidden_size,layer2_hidden_size)
self.w3 = np.random.randn(layer2_hidden_size,num_nn_output)
```

3. Back propagation：

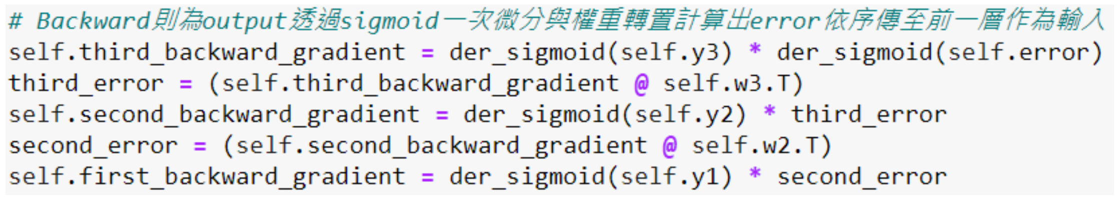
   In order to complete back propagation, it’s essential to implement forward function before backward. So first we need to calculate the value of output layer by layer.

   The flow is, given inputs passing through (by multiply) weights and adding bias (there’s no need in this lab) and getting outputs. And we feed these outputs as second layer’s inputs, layer by layer, to calculate last output.



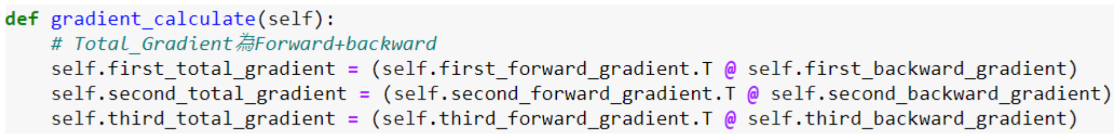
```python
# Forward為input經過weight傳過sigmoid成為下一層的input，一層一層傳至最後一層輸出output
self.first_forward_gradient = inputs
y1 = sigmoid(inputs @ self.w1)
self.y1 = y1

self.second_forward_gradient = y1
y2 = sigmoid(y1 @ self.w2)
self.y2 = y2

self.third_forward_gradient = y2
y3 = sigmoid(y2 @ self.w3)
self.y3 = y3
return y3.T
```

This output is reversely fed into the derivation of sigmoid function, multiplying the transpose of weights matrix, to calculate “error”.



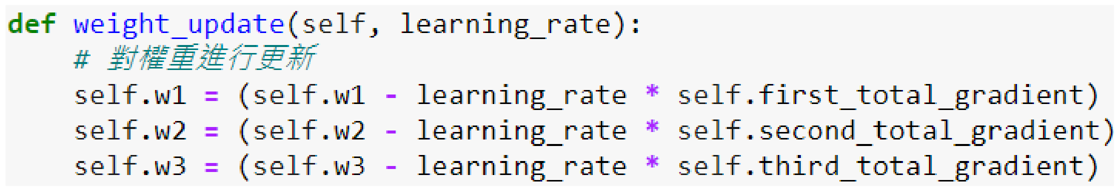
```python
# Backward則為output透過sigmoid一次微分與權重轉置計算出error依序傳至前一層作為輸入
self.third_backward_gradient = der_sigmoid(self.y3) * der_sigmoid(self.error)
third_error = (self.third_backward_gradient @ self.w3.T)
self.second_backward_gradient = der_sigmoid(self.y2) * third_error
second_error = (self.second_backward_gradient @ self.w2.T)
self.first_backward_gradient = der_sigmoid(self.y1) * second_error
```

Then, we can get total gradient by adding forward and backward these two functions.



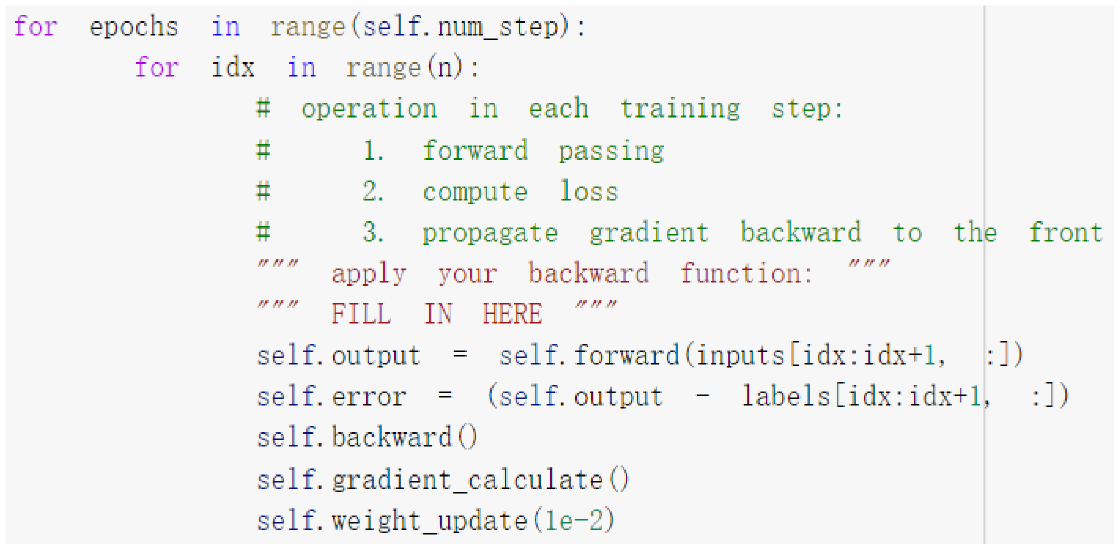
```python
def gradient_calculate(self):
    # Total_Gradient為Forward+backward
    self.first_total_gradient = (self.first_forward_gradient.T @ self.first_backward_gradient)
    self.second_total_gradient = (self.second_forward_gradient.T @ self.second_backward_gradient)
    self.third_total_gradient = (self.third_forward_gradient.T @ self.third_backward_gradient)
```

After that, we should update each weights to complete one propagation.



```python
def weight_update(self, learning_rate):
    # 對權重進行更新
    self.w1 = (self.w1 - learning_rate * self.first_total_gradient)
    self.w2 = (self.w2 - learning_rate * self.second_total_gradient)
    self.w3 = (self.w3 - learning_rate * self.third_total_gradient)
```

Finally, adding these functions into epoch loops of “train” functions to implement a complete neural network model.



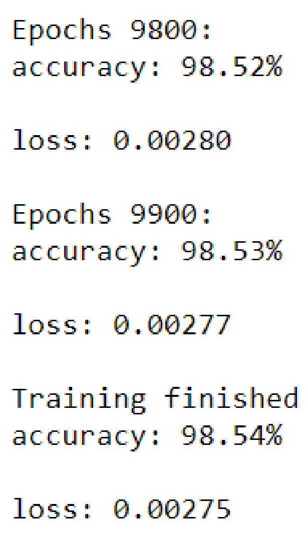
```python
for epochs in range(self.num_step):
    for idx in range(n):
        # operation in each training step:
        #   1. forward passing
        #   2. compute loss
        #   3. propagate gradient backward to the front
        """ apply your backward function: """
        """ FILL IN HERE """
        self.output = self.forward(inputs[idx:idx+1, :])
        self.error = (self.output - labels[idx:idx+1, :])
        self.backward()
        self.gradient_calculate()
        self.weight_update(1e-2)
```

III. Experimental Result

a. Specify training and results

In “linear”, 10000 steps of run are used for a batch size of 100, that means there’s 100 epochs exist.

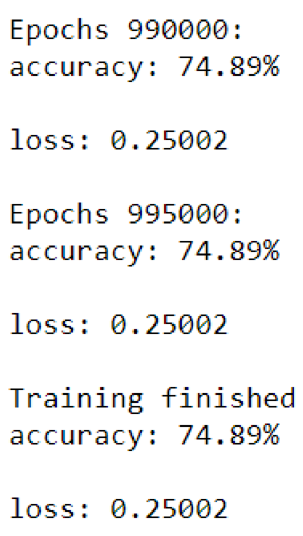
Below figure is last three epochs, with a mse final loss 0.00275.



```
Epochs 9800:
accuracy: 98.52%

loss: 0.00280

Epochs 9900:
accuracy: 98.53%

loss: 0.00277

Training finished
accuracy: 98.54%

loss: 0.00275
```

Relatively, “XOR” run 1000000 steps and batch size 5000(200 epochs)

Below figure is last three epochs, with a mse final loss 0.25002.

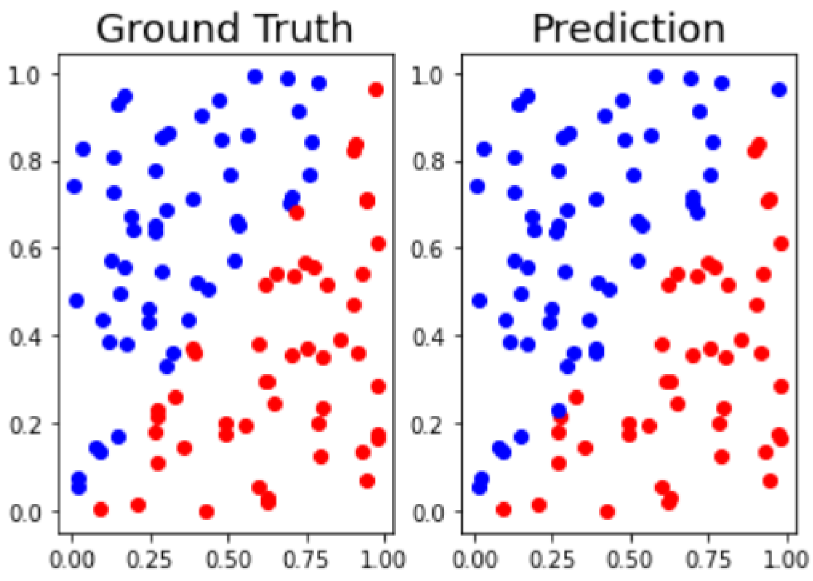


```
Epochs 990000:
accuracy: 74.89%

loss: 0.25002

Epochs 995000:
accuracy: 74.89%

loss: 0.25002

Training finished
accuracy: 74.89%

loss: 0.25002
```

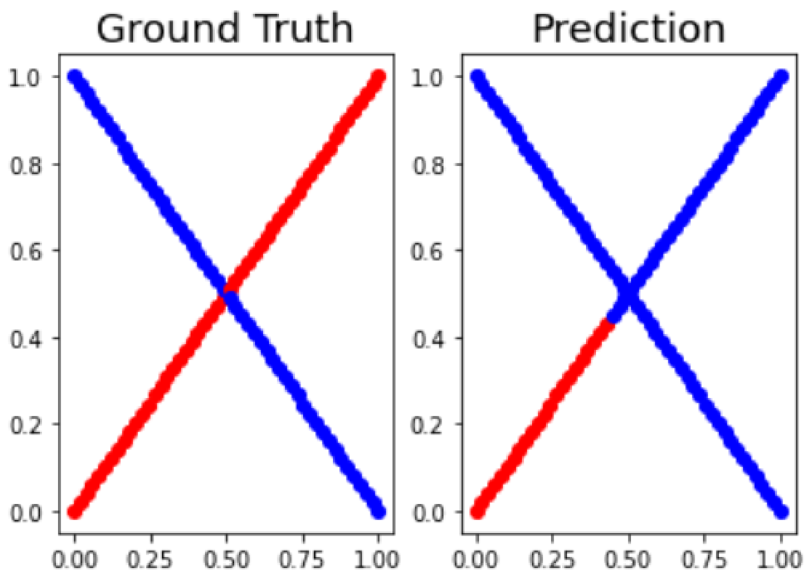
b. Figures

Parts of loss and prediction screenshots are shown above, so in this part I only paste the comparison plot.

Linear:



XOR:



IV. Discussion and extra experiments

I have try making trains more steps, but accuracy grows slowly when it runs dozens of epochs. I think it may have stuck in local optimum. This condition happens especially with XOR function, accuracy difference is quite small between 50000 steps and 1000000 steps.