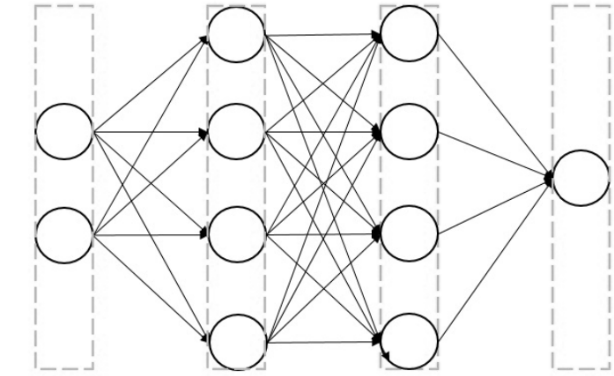
Deep Learning and Practice Lab1 - back-propagation

310554009 杜葳葳

1. Introduction

I implement a two hidden layers neural networks without calling function. Generate two kinds of data, linear and XOR, in order to test the network.

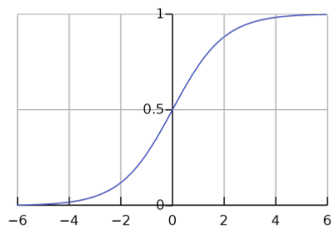


2. Experiment setups

A. Sigmoid functions

- Definition: A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point.
- Purpose: It produces output in scale of [0, 1], and its derivative is easy to demonstrate.
- Formula:

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

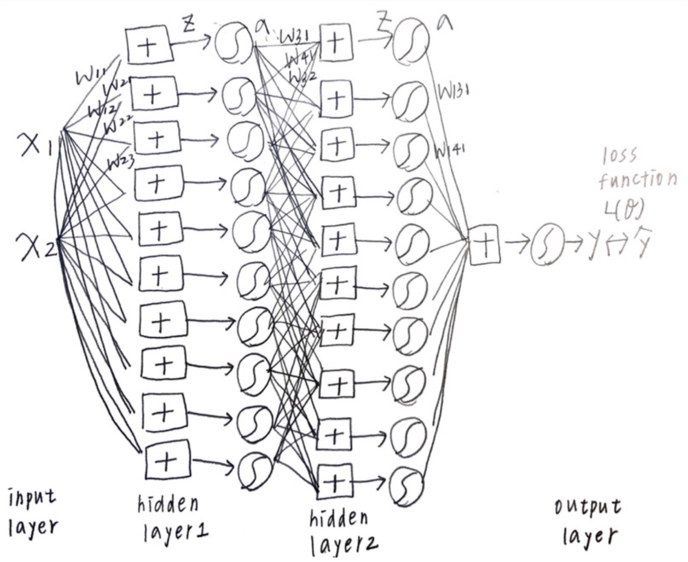


- Implementation:

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

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

B. Neural network



Input: 2 dimension

Output: 1 dimension

Number of hidden layer: 2

Number of epoch: 10000

Loss function: MSE

C. Backpropagation

a. Overall

generate data → train() → forward() → loss → backpropagation() → draw the plot

```python
def train(x,y):
    ## init network weights
    weight1 = np.random.random((input_dim, hidden_layer_dim))
    weight2 = np.random.random((hidden_layer_dim, hidden_layer_dim))
    weight3 = np.random.random((hidden_layer_dim, output_dim))
    epoch_list = [] # for plot
    loss_list = [] # for plot

    ## for each training example
    for i in range(num_epoch):
        loss = 0
        for j in range(len(x)):
            ## forward pass
            pred, outputs = forward(weight1, weight2, weight3, x[j].reshape(1, 2))
            ## loss
            loss += loss_function(pred, y[j].reshape(1, 1))
            ## backpropagation
            weight1, weight2, weight3 = backpropagation(x[j].reshape(1,
                2),y[j].reshape(1, 1), outputs, weight1,weight2,weight3)
        epoch_list.append(i)
        loss_list.append(loss)
        if (i+1) % 100 == 0:
            print("epoch {} loss : {}".format(i+1, loss))

    pred, _ = forward(weight1, weight2, weight3, x)
    print("Prediction: ", pred)

    num_correct = 0
    for i in range(len(pred)):
        if pred[i] > 0.5:
            pred[i] = 1
        else:
            pred[i] = 0
        if pred[i]==y[i]:
            num_correct +=1
    print("Accuracy: ", num_correct / len(pred))

    show_result(x, y, pred)
    show_curve(epoch_list, loss_list)

    return weight1, weight2, weight3
```

```python
def forward(weight1,weight2,weight3,x):
    a_list = []

    a = sigmoid(np.matmul(x, weight1))
    a_list.append(a)
    a = sigmoid(np.matmul(a, weight2))
    a_list.append(a)
    a = sigmoid(np.matmul(a, weight3))
    a_list.append(a)

    return a, a_list
```

```python
def backpropagation(x, y, outputs, weight1, weight2, weight3):
    ## output layer
    a2_z2 = derivative_sigmoid(outputs[2])
    c_a2 = 2 * (outputs[2] - y)
    c_z2 = np.matmul(a2_z2, c_a2)
    c_w2 = np.matmul(outputs[1].T, c_z2)

    ## layer 2
    for i in range(len(weight3)):
        a1_z1 = np.array([derivative_sigmoid(outputs[1]).reshape(-1)[i]]).reshape(1, 1)
        z2_a1 = np.array([weight3[i]])

        c_a1 = np.matmul(z2_a1, c_z2)
        z1_w1 = outputs[0].T
        c_z1_notconcat = np.matmul(a1_z1, c_a1)

        if i == 0:
            c_z1 = np.matmul(a1_z1, c_a1)
            c_w1 = np.matmul(z1_w1, c_z1)
        else:
            c_z1 = np.concatenate((c_z1, np.matmul(a1_z1, c_a1)), axis = 1)
            c_w1 = np.concatenate((c_w1, np.matmul(z1_w1, c_z1_notconcat)), axis = 1)

    ## layer 1
    for i in range(len(weight2)):
        a0_z0 = np.array([derivative_sigmoid(outputs[0]).reshape(-1)[i]]).reshape(1, 1)
        z1_a0 = np.array([weight2[i]]).reshape(len(weight2), 1)

        c_a0 = np.matmul(z1_a0, c_z1)
        z0_w0 = x.T
        c_a0 = np.matmul(z1_a0.T, c_z1.T)
        c_z0 = np.matmul(a0_z0, c_a0)
        if i == 0:
            c_w0 = np.matmul(z0_w0, c_z0)
        else:
            c_w0 = np.concatenate((c_w0, np.matmul(z0_w0, c_z0)), axis = 1)

    weight1 -= learning_rate * c_w0
    weight2 -= learning_rate * c_w1
    weight3 -= learning_rate * c_w2
    return weight1, weight2, weight3
```

b. Backpropagation

randomly set the weight → for each training example (forward → loss → backward → update weight) → get the optimized weight

$$\frac{\partial C}{\partial w} = \underbrace{\boxed{\frac{\partial z}{\partial w}}}_{\text{forward path}} \overbrace{\boxed{\frac{\partial C}{\partial z}}}^{\text{backward path}}$$

$$\frac{\partial C}{\partial z} = \underbrace{\boxed{\frac{\partial a}{\partial z}}}_{\sigma'(z)} \boxed{\frac{\partial C}{\partial a}} \rightarrow \frac{\partial C}{\partial a} = \underbrace{\frac{\partial z'}{\partial a}}_{\text{weight}} \frac{\partial C}{\partial z'} + \underbrace{\frac{\partial z''}{\partial a}}_{\text{weight}} \frac{\partial C}{\partial z''} + \cdots \quad \text{chain rule}$$

case 1: output layer

$$\frac{\partial C}{\partial z'} = \frac{\partial y}{\partial z'} \boxed{\frac{\partial C}{\partial y}}$$

$$\hookrightarrow \frac{\partial C}{\partial y} = \frac{\partial (y-\hat{y})^2}{\partial y} = 2(y-\hat{y})$$

case 2: not output layer

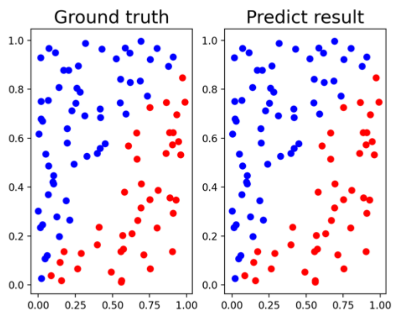
$$\frac{\partial C}{\partial a} = \frac{\partial z'}{\partial a} \times \frac{\partial C}{\partial z'} + \frac{\partial z''}{\partial a} \times \frac{\partial C}{\partial z''} + \cdots$$

$\hookrightarrow$ send from next layer

3. Results of your testing

A. Screenshot and comparison figure

- Linear

# Epoch = 10000, Learning rate = 0.1 Hidden layer dimension = 10		
Training: loss values	Testing: predictions	
epoch 100 loss : 24.564707621503068 epoch 200 loss : 24.5427961400792 epoch 300 loss : 24.507245322909057 epoch 400 loss : 24.43882699454884 epoch 500 loss : 24.27484277529711 epoch 600 loss : 23.741386366198867 epoch 700 loss : 21.05265936536282 epoch 800 loss : 10.99447421390158 epoch 900 loss : 5.297157906561916 epoch 1000 loss : 3.538659546113353 epoch 9000 loss : 0.08179285321049261 epoch 9100 loss : 0.08008622824093589 epoch 9200 loss : 0.07843645990794493 epoch 9300 loss : 0.07684109000918622 epoch 9400 loss : 0.07529778597043751 epoch 9500 loss : 0.07380433368582484 epoch 9600 loss : 0.07235863079333543 epoch 9700 loss : 0.07095868035851834 epoch 9800 loss : 0.06960258494083052 epoch 9900 loss : 0.06828854101874225 epoch 10000 loss : 0.06701483375114707	Prediction: [[9.99133987e-01] [9.99111426e-01] [2.53102133e-02] [9.98748669e-01] [9.98558585e-01] [9.99084447e-01] [9.99306349e-01] [9.99287210e-01] [9.98547426e-01] [9.92789584e-01] [9.98804052e-01] [9.99198632e-01] [8.88781235e-01]	

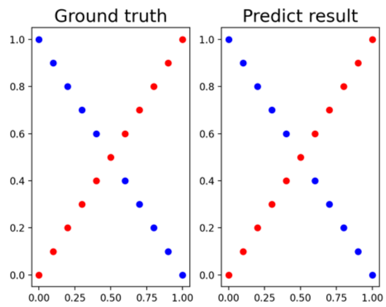
- XOR

# Epoch = 10000, Learning rate = 0.1 Hidden layer dimension = 10		
Training: loss values	Testing: predictions	

```
epoch 100 loss : 5.880815786782352
epoch 200 loss : 5.870709121848083
epoch 300 loss : 5.857698975161231
epoch 400 loss : 5.840873606671649
epoch 500 loss : 5.819657621597579
epoch 600 loss : 5.794149882779938
epoch 700 loss : 5.764352590970916
epoch 800 loss : 5.729665567216511
epoch 900 loss : 5.6901640074061675
epoch 1000 loss : 5.647740819224648
```

```
epoch 9000 loss : 0.0030135135883186956
epoch 9100 loss : 0.002946711780308359
epoch 9200 loss : 0.0028825634187972095
epoch 9300 loss : 0.002820919125538334
epoch 9400 loss : 0.0027616402983024523
epoch 9500 loss : 0.002704598169514072
epoch 9600 loss : 0.0026496729609213816
epoch 9700 loss : 0.0025967531231435577
epoch 9800 loss : 0.0025457346503900994
epoch 9900 loss : 0.0024965204618849265
epoch 10000 loss : 0.0024490198425966077
```

```
Prediction:  [[2.69488942e-05]
 [9.99868366e-01]
 [1.58456194e-04]
 [9.99882262e-01]
 [1.26061646e-03]
 [9.99890651e-01]
 [7.77217707e-03]
 [9.99826685e-01]
 [2.02853292e-02]
 [9.76040894e-01]
 [2.18342560e-02]
 [1.31463808e-02]
 [9.73823670e-01]
 [6.05174313e-03]
 [9.99898444e-01]
 [2.66057530e-03]
 [9.99972923e-01]
 [1.25828112e-03]
 [9.99981741e-01]
 [6.68387089e-04]
 [9.99984094e-01]]
```



B. Show the accuracy of your prediction

- Linear

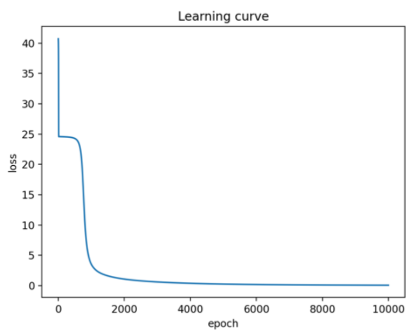
```
Accuracy:  1.0
```

- XOR

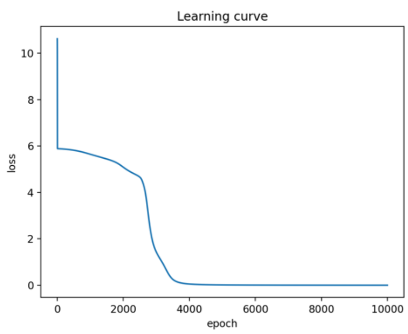
```
Accuracy:  1.0
```

C. Learning curve (Loss, Epoch curve)

- Linear



- XOR



D. Anything you want to present

- Time: more hidden layers → cost more time

4. Discussion

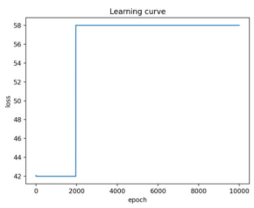
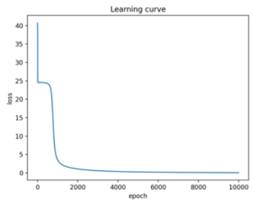
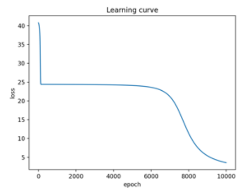
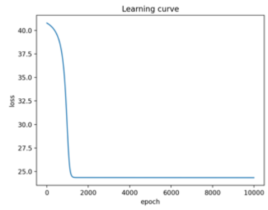
A. Try different **learning curve**

We have to set appropriate learning rate. The learning rate controls how quickly the model is adapted to the problem. Too small may result in a long training process that could get stuck, however, too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

a. Linear

After testing, learning rate in the range between 0.1 to 0.001 is a suitable learning rate. The results are as follows.

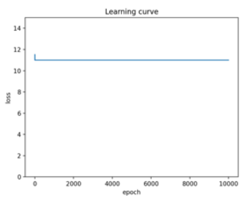
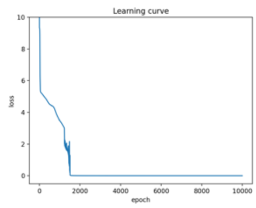
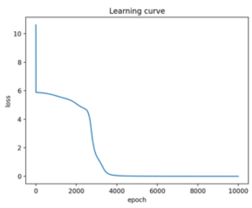
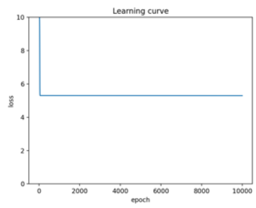
Learning rate	**10**	**0.1**	**0.001**	**0.0001**
Accuracy	0.42	1.0	1.0	0.58

Learning rate=10	Learning rate=0.1	Learning rate=0.001	Learning rate=0.0001
			

b. XOR

After testing, learning rate in the range between 1 to 0.01 is a suitable learning rate. The results are as follows.

Learning rate	**10**	**1**	**0.1**	**0.01**
Accuracy	0.58	1.0	1.0	0.53

Learning rate=10	Learning rate=1	Learning rate=0.1	Learning rate=0.01
			

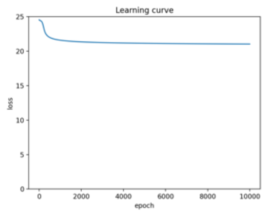
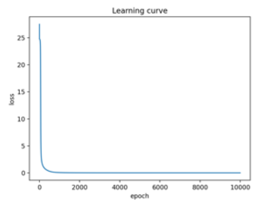
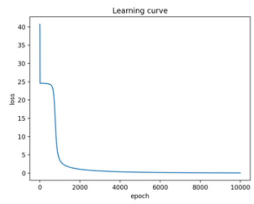
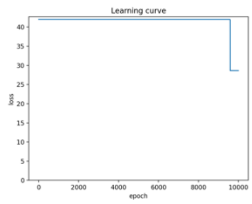
B. Try different **numbers of hidden units**

More hidden units, higher model complexity. There is no rule for that. A trial-and-error process has been used.

a. Linear

After testing, numbers of hidden units in the range between 5 to 10 is a suitable number. The results are as follows.

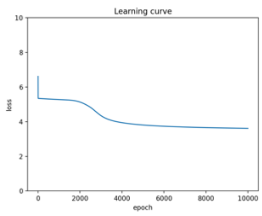
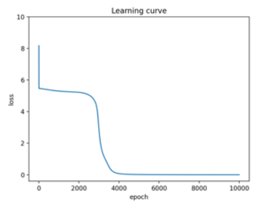
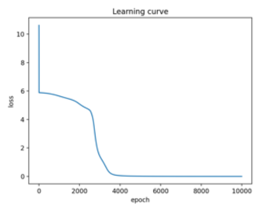
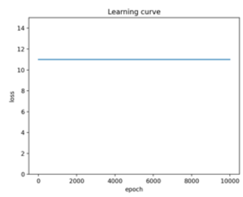
Hidden units	**1**	**5**	**10**	**30**
Accuracy	0.58	1.0	1.0	0.58

Hidden units = 1	Hidden units = 5	Hidden units = 10	Hidden units = 30
			

b. XOR

After testing, numbers of hidden units in the range between 5 to 10 is a suitable number. The results are as follows.

Hidden units	**3**	**5**	**10**	**100**
Accuracy	0.76	1.0	1.0	0.48

Hidden units = 3	Hidden units = 5	Hidden units = 10	Hidden units = 100
			

C. Try without **activation functions**

A neural network without an activation function is essentially just a linear regression model. As a result, the result cannot converge.

a. Linear

```
epoch 100 loss : nan
epoch 200 loss : nan
epoch 300 loss : nan
epoch 400 loss : nan
epoch 500 loss : nan
epoch 600 loss : nan
epoch 700 loss : nan
epoch 800 loss : nan
epoch 900 loss : nan
epoch 1000 loss : nan
```

```
epoch 9000 loss : nan
epoch 9100 loss : nan
epoch 9200 loss : nan
epoch 9300 loss : nan
epoch 9400 loss : nan
epoch 9500 loss : nan
epoch 9600 loss : nan
epoch 9700 loss : nan
epoch 9800 loss : nan
epoch 9900 loss : nan
epoch 10000 loss : nan
```

```
Prediction:  [[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
```

b. XOR

```
epoch 100 loss : nan
epoch 200 loss : nan
epoch 300 loss : nan
epoch 400 loss : nan
epoch 500 loss : nan
epoch 600 loss : nan
epoch 700 loss : nan
epoch 800 loss : nan
epoch 900 loss : nan
epoch 1000 loss : nan
```

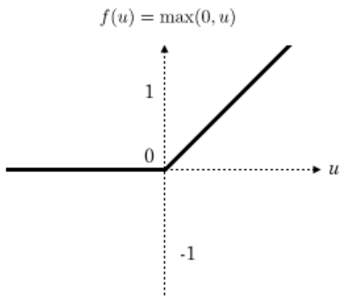
```
epoch 9000 loss : nan
epoch 9100 loss : nan
epoch 9200 loss : nan
epoch 9300 loss : nan
epoch 9400 loss : nan
epoch 9500 loss : nan
epoch 9600 loss : nan
epoch 9700 loss : nan
epoch 9800 loss : nan
epoch 9900 loss : nan
epoch 10000 loss : nan
```

```
Prediction:  [[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]
```

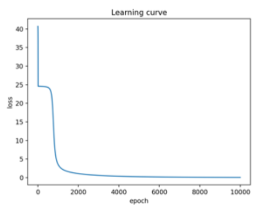
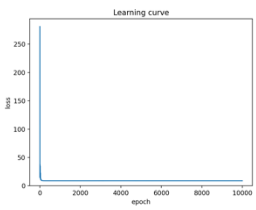
D. Anything you want to share

a. Different activation function: Linear

Try ReLU as the activation function of the network. The accuracy of linear data is 0.9, less than Sigmoid.



Activation function	**Sigmoid**	**ReLU**
Accuracy	1.0	0.9

Sigmoid	**ReLU**
	

b. XOR

Try ReLU as the activation function of the network. The accuracy of linear data is 1.0, same as Sigmoid, but the learning curve is not stable.

Activation function	**Sigmoid**	**ReLU**
Accuracy	1.0	1.0

Sigmoid	**ReLU**