

Reading Circle #5

Chapter5 Backward Propagation Neural Network

2018/06/18

Zhongjie Zhang

D1

5.6 Realization of Affine and Softmax Layer

In the backward propagation network model, the propagation can be decomposed into several layers. The function of propagation is mainly rely on the affine layers. And in this example, we use softmax layer as the output layer.

5.6.1 Affine Layer

The parameters based on former node calculation are all numbers.

We apply Matrix multiplication to the calculation of the Neural Networks.

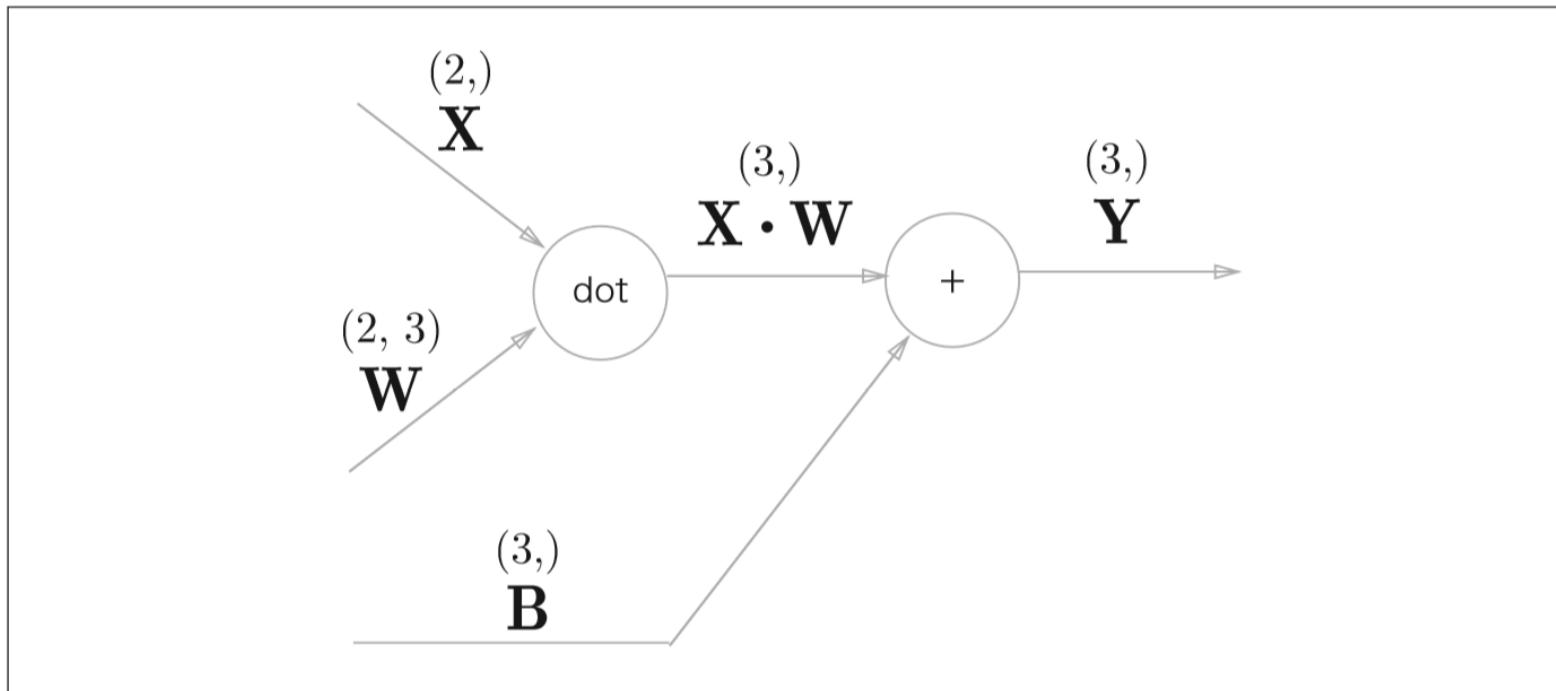
$$Y = \text{np.dot}(X, W) + B$$

So we can apply the Matrix multiplication to layer calculation and we call the layer as affine layer.

5.6.1 Affine Layer

We must pay attention to the rule of the Matrix multiplication that we must match the rows and columns.

$$X(1,2) * W(2,3) + b(1,3) = Y(1,3)$$



5.6.1 Affine Layer

Expression of Partial derivative

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T$$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \cdot \frac{\partial L}{\partial \mathbf{Y}}$$

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\mathbf{W}^T = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}$$

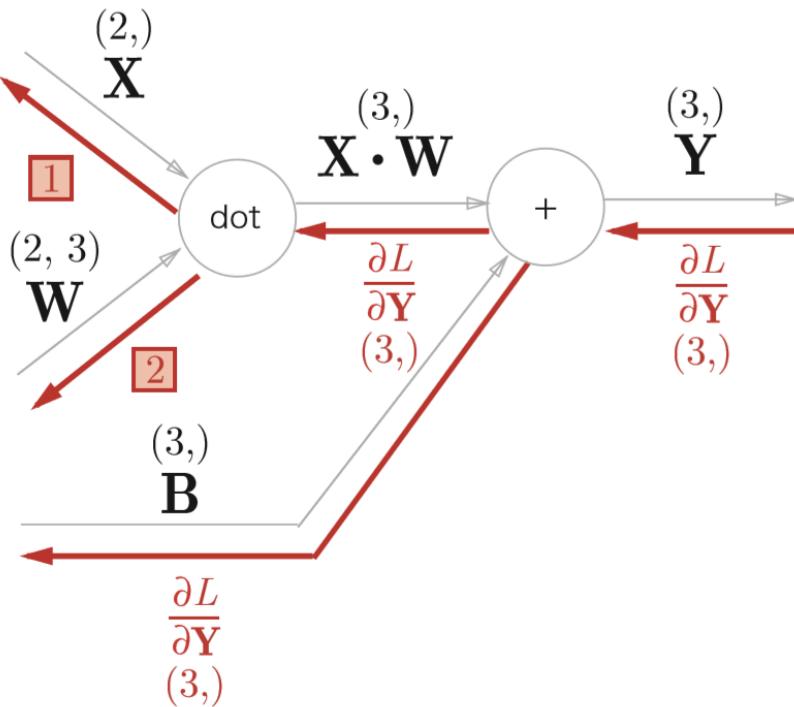
T: Matrix transpose

5.6.1 Affine Layer

We also need to pay attention to the rows and columns whether match or not in the backward propagation.

$$1 \quad \frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T$$
$$(2,) \quad (3,) \quad (3, 2)$$

$$2 \quad \frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \cdot \frac{\partial L}{\partial \mathbf{Y}}$$
$$(2, 3) \quad (2, 1) \quad (1, 3)$$

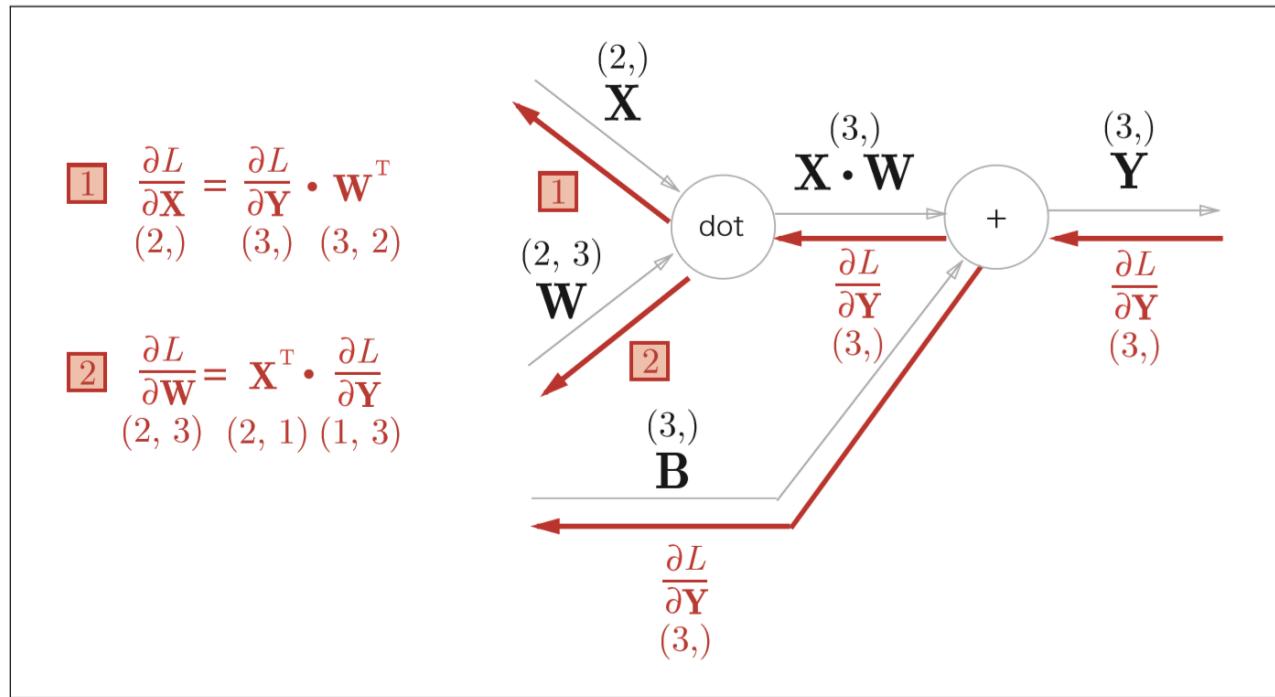


5.6.1 Affine Layer

Attention to shape

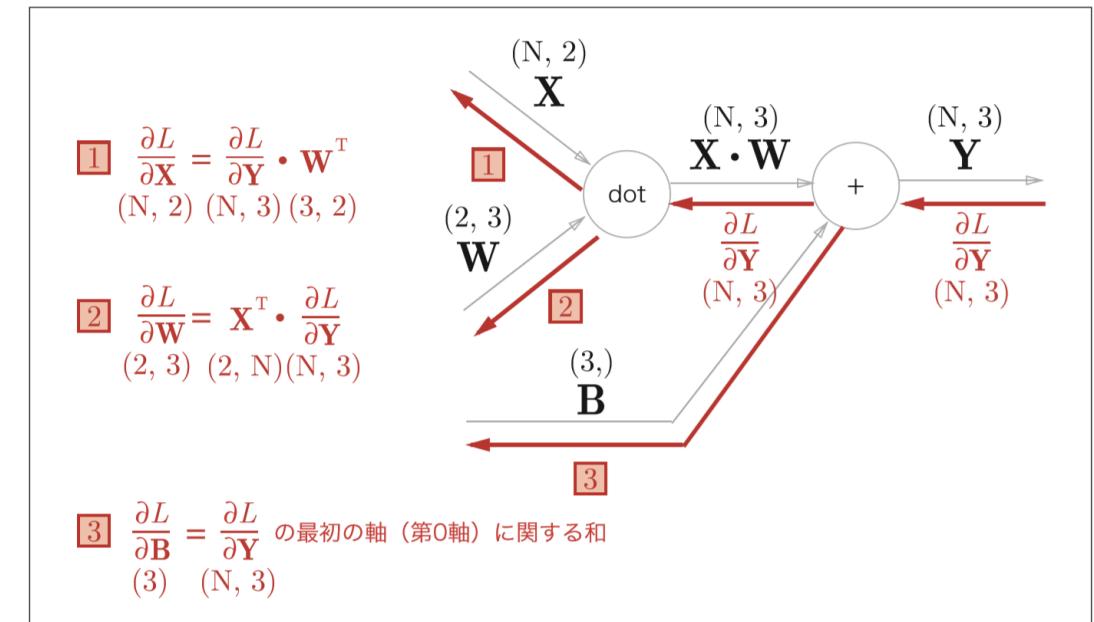
$$\mathbf{X} = (x_0, x_1, \dots, x_n)$$

$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \dots, \frac{\partial L}{\partial x_n} \right)$$



5.6.2 Batch Affine Layer

The principle is like Chapter 4. Assuming the data is a (1,M)Matrix, then the (N*M)matrix means that there are N datas input at the same time. This batch process can improve the efficiency of the calculation.

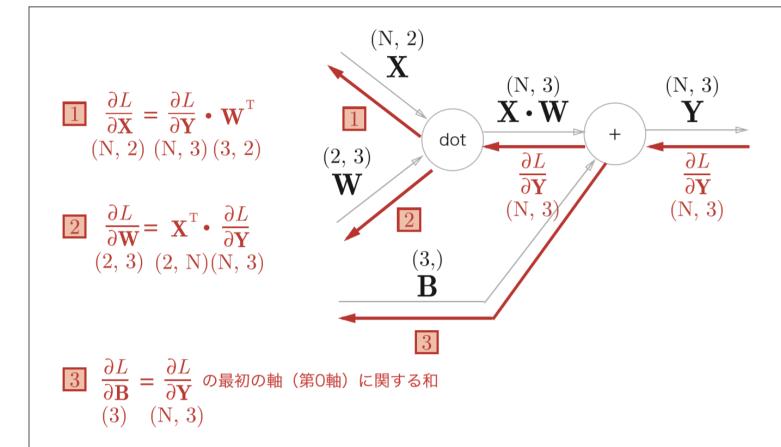


5.6.2 Batch Affine Layer

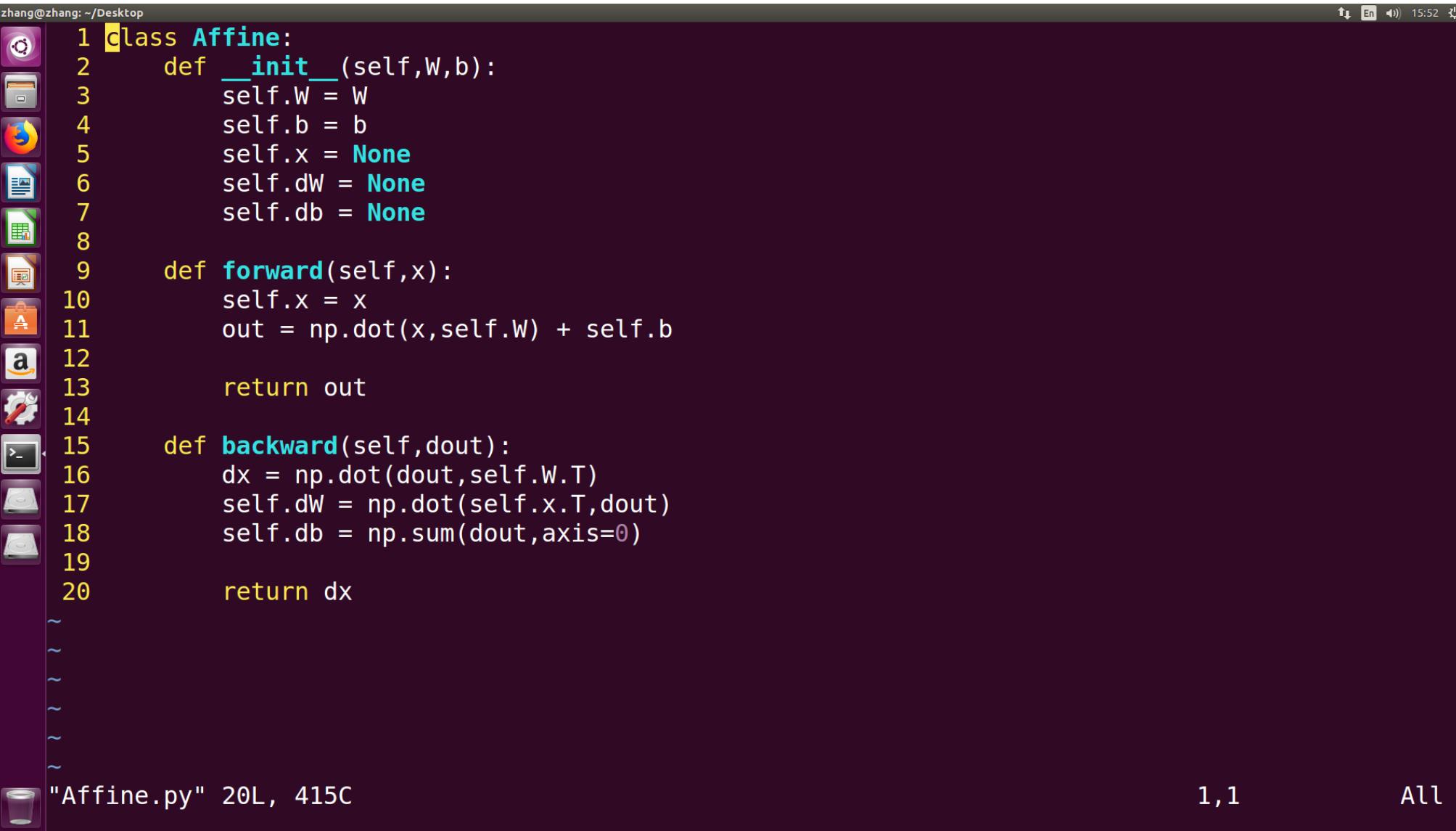
About the bias

We add the bias to all the data in the forward propagation. So we need to add all the bias in rows.

```
>>> X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])      >>> dY = np.array([[1, 2, 3], [4, 5, 6]])
>>> B = np.array([1, 2, 3])                            >>> dY
>>>
>>> X_dot_W                                         array([[1, 2, 3],
array([[ 0,  0,  0],           [4, 5, 6]])
   [ 10, 10, 10]])
>>> X_dot_W + B                                     >>>
array([[ 1,  2,  3],                         >>> dB = np.sum(dY, axis=0)
   [11, 12, 13]])                                >>> dB
                                                               array([5, 7, 9])
```



5.6.2 Batch Affine Layer



A screenshot of a Linux desktop environment, specifically Ubuntu, showing a terminal window. The terminal window has a dark purple background and displays the following Python code:

```
zhang@zhang: ~/Desktop
1 class Affine:
2     def __init__(self,W,b):
3         self.W = W
4         self.b = b
5         self.x = None
6         self.dW = None
7         self.db = None
8
9     def forward(self,x):
10        self.x = x
11        out = np.dot(x,self.W) + self.b
12
13        return out
14
15    def backward(self,dout):
16        dx = np.dot(dout,self.W.T)
17        self.dW = np.dot(self.x.T,dout)
18        self.db = np.sum(dout, axis=0)
19
20        return dx
~
~
```

The code defines a class named `Affine` with two methods: `forward` and `backward`. The `forward` method performs the affine transformation $y = \mathbf{w}^T \mathbf{x} + b$ and stores the input \mathbf{x} and the transformed output y for backpropagation. The `backward` method calculates the gradients of the loss function with respect to the input \mathbf{x} and the weights \mathbf{w} , and updates the gradients stored in `dW` and `db`.

At the bottom of the terminal window, it shows the file name "Affine.py" and its status: 20L, 415C. The bottom right corner also shows the number 1,1 and the word All.

5.6.3 Softmax-with-Loss Layer

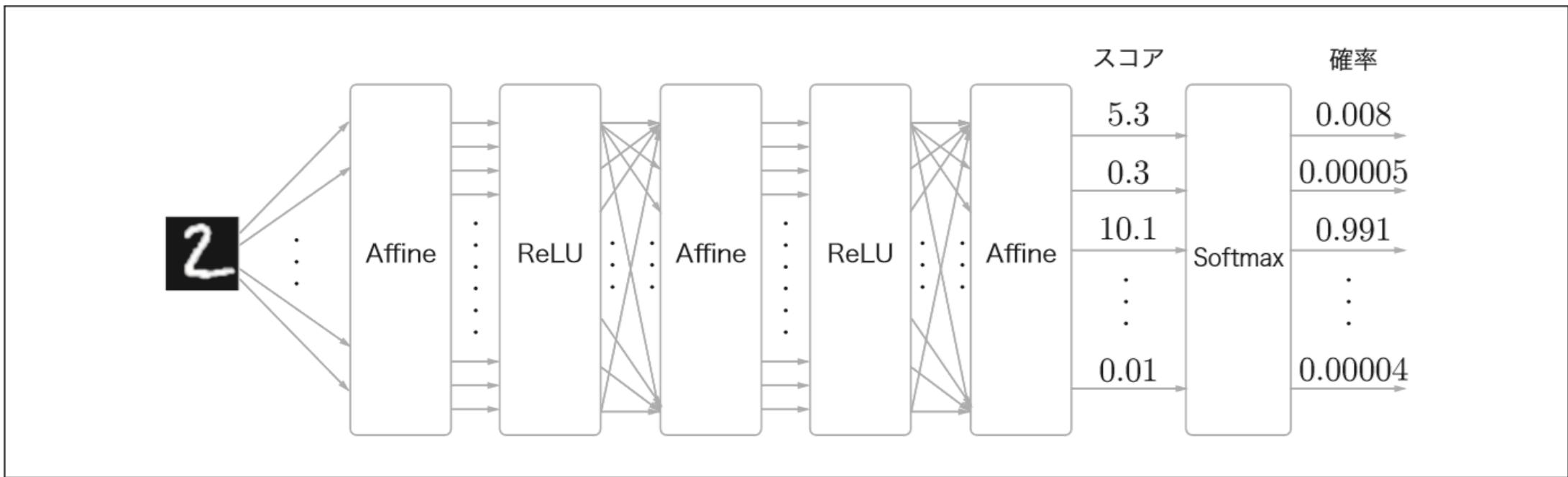
Softmax Function can make the value of the output normalized which means the sum of all the output's value is 1.

According to the monotonicity of the softmax function,we can use the value of output before the softmax to do the inference.

And we call the value as score.

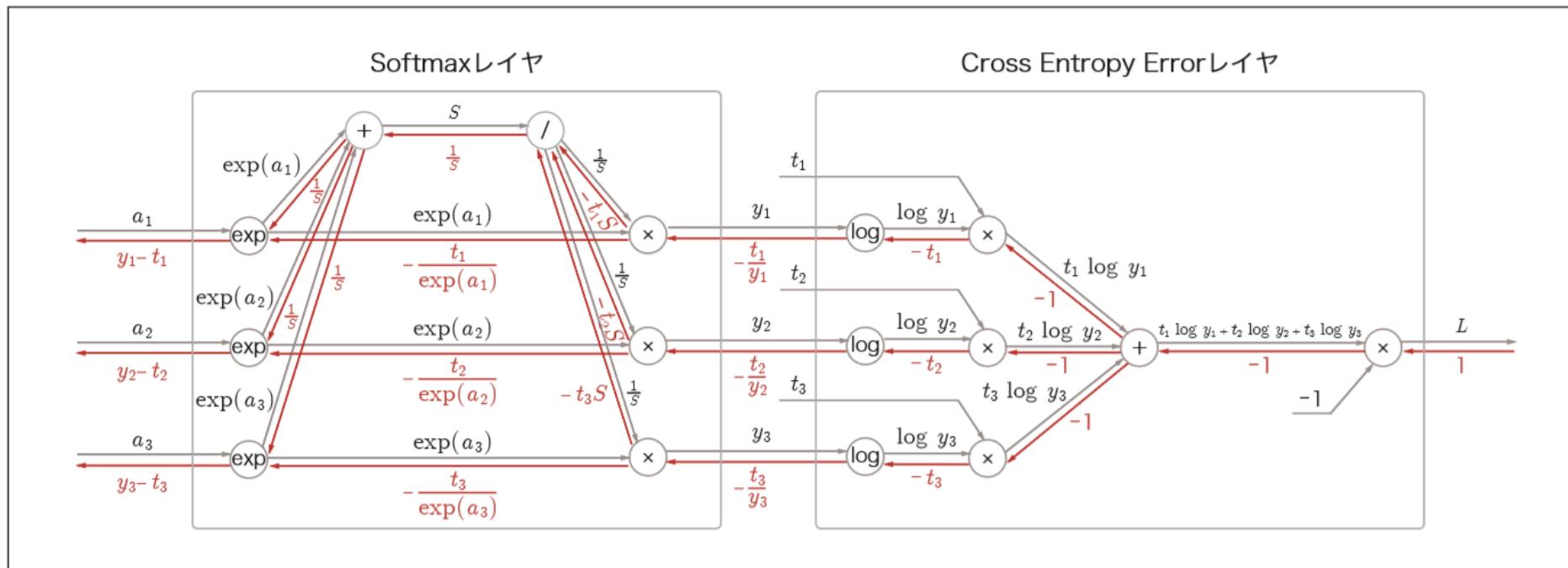
But if we need the data to do the learning,we should use the softmax function to make all the value normalized.

5.6.3 Softmax-with-Loss Layer



5.6.3 Softmax-with-Loss Layer

In order to make learning more easier, we put the output function and loss function together. We use softmax as output and cross entropy error as loss function.



5.6.3 Softmax-with-Loss Layer

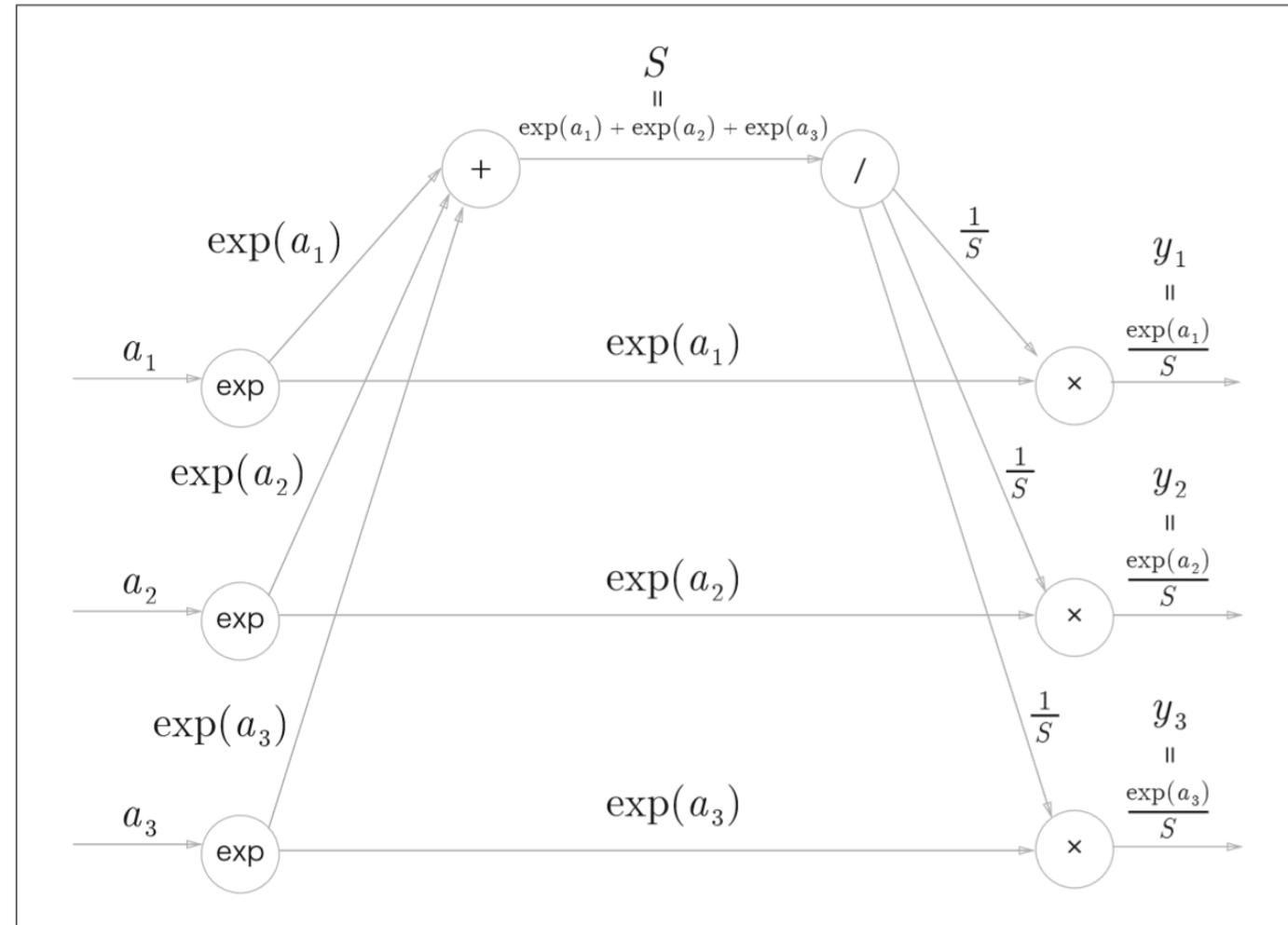
Forward Propagation

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^n \exp(a_i)}$$

$$L = - \sum_k t_k \log y_k$$

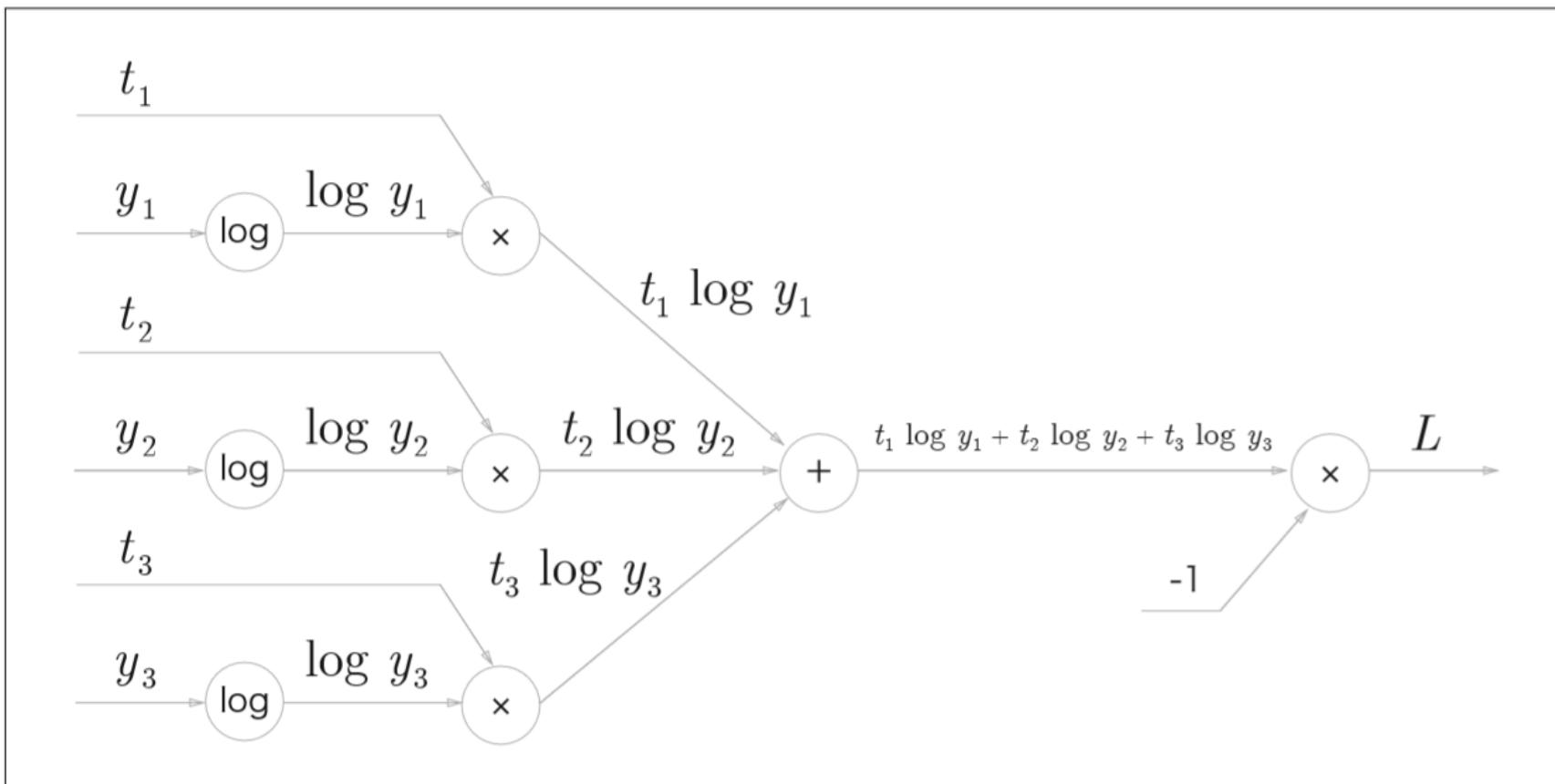
5.6.3 Softmax-with-Loss Layer

Forward Propagation



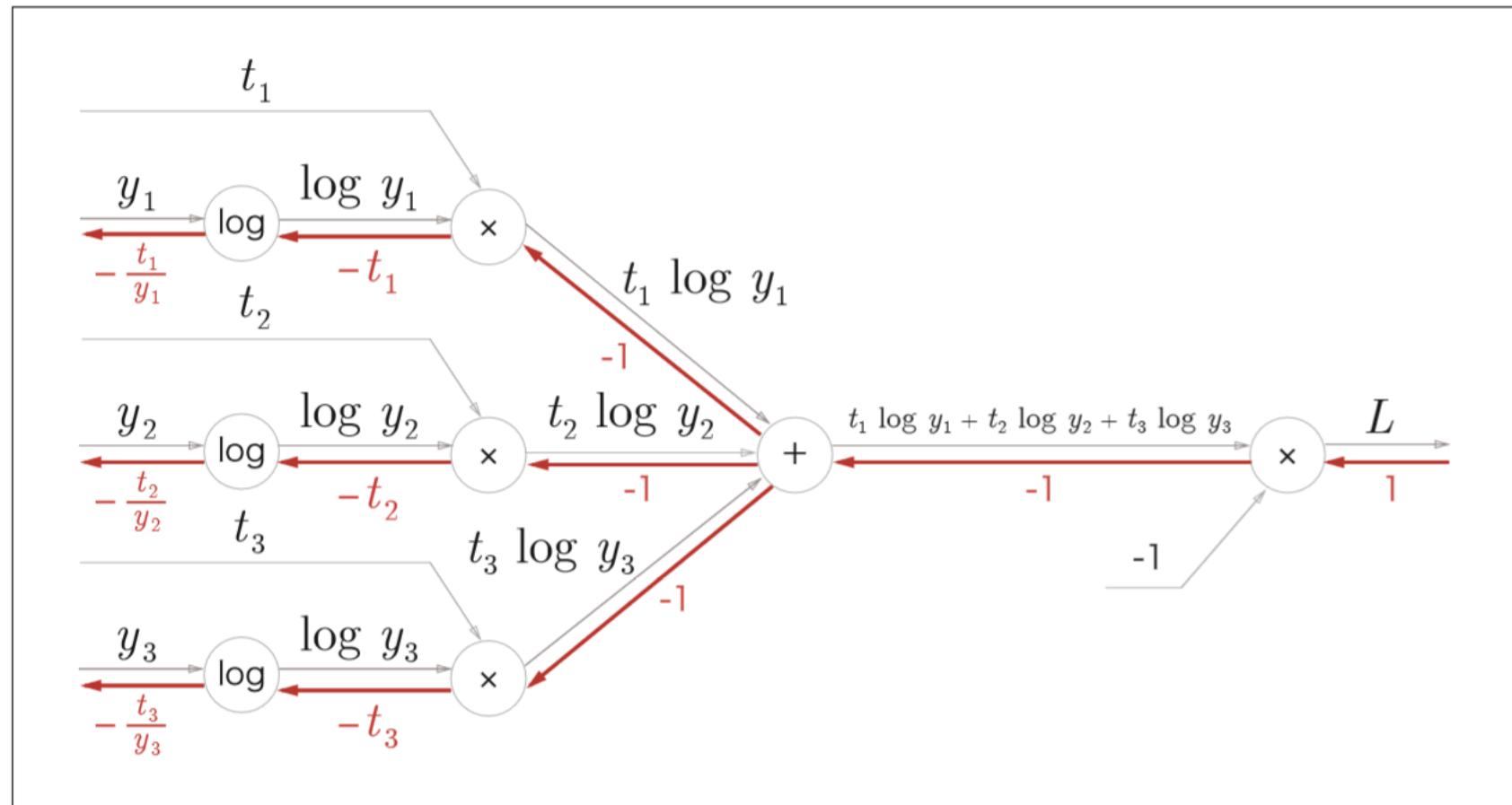
5.6.3 Softmax-with-Loss Layer

Forward Propagation



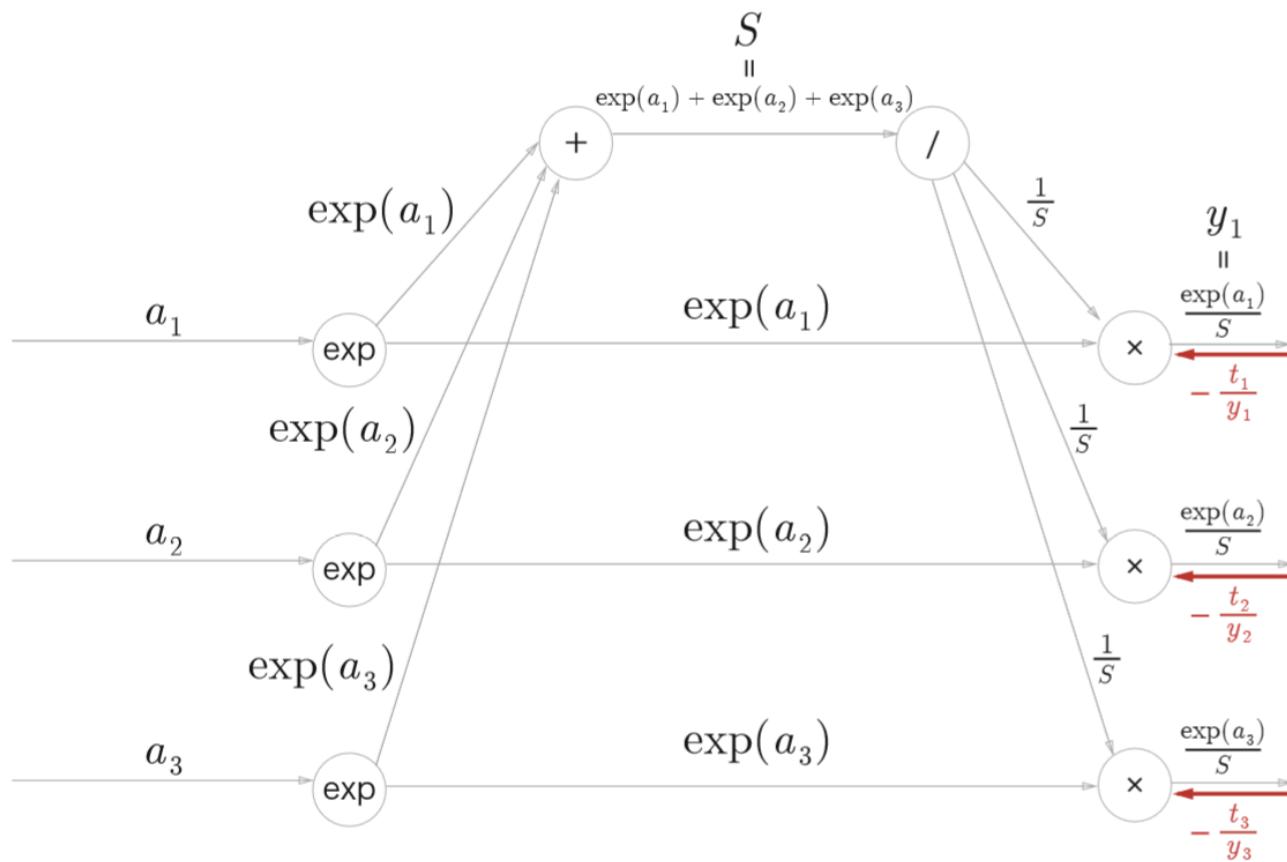
5.6.3 Softmax-with-Loss Layer

Backward propagation



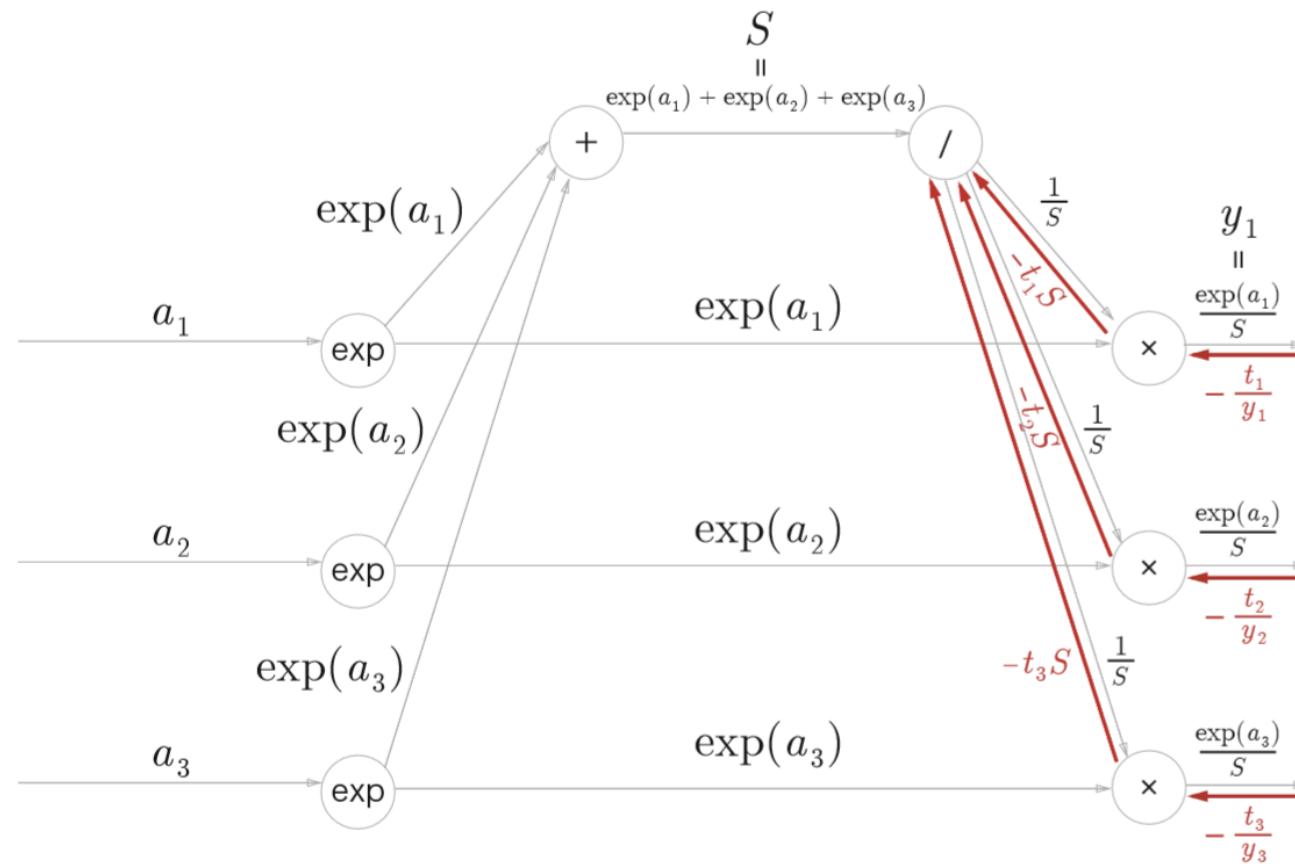
5.6.3 Softmax-with-Loss Layer

Backward propagation



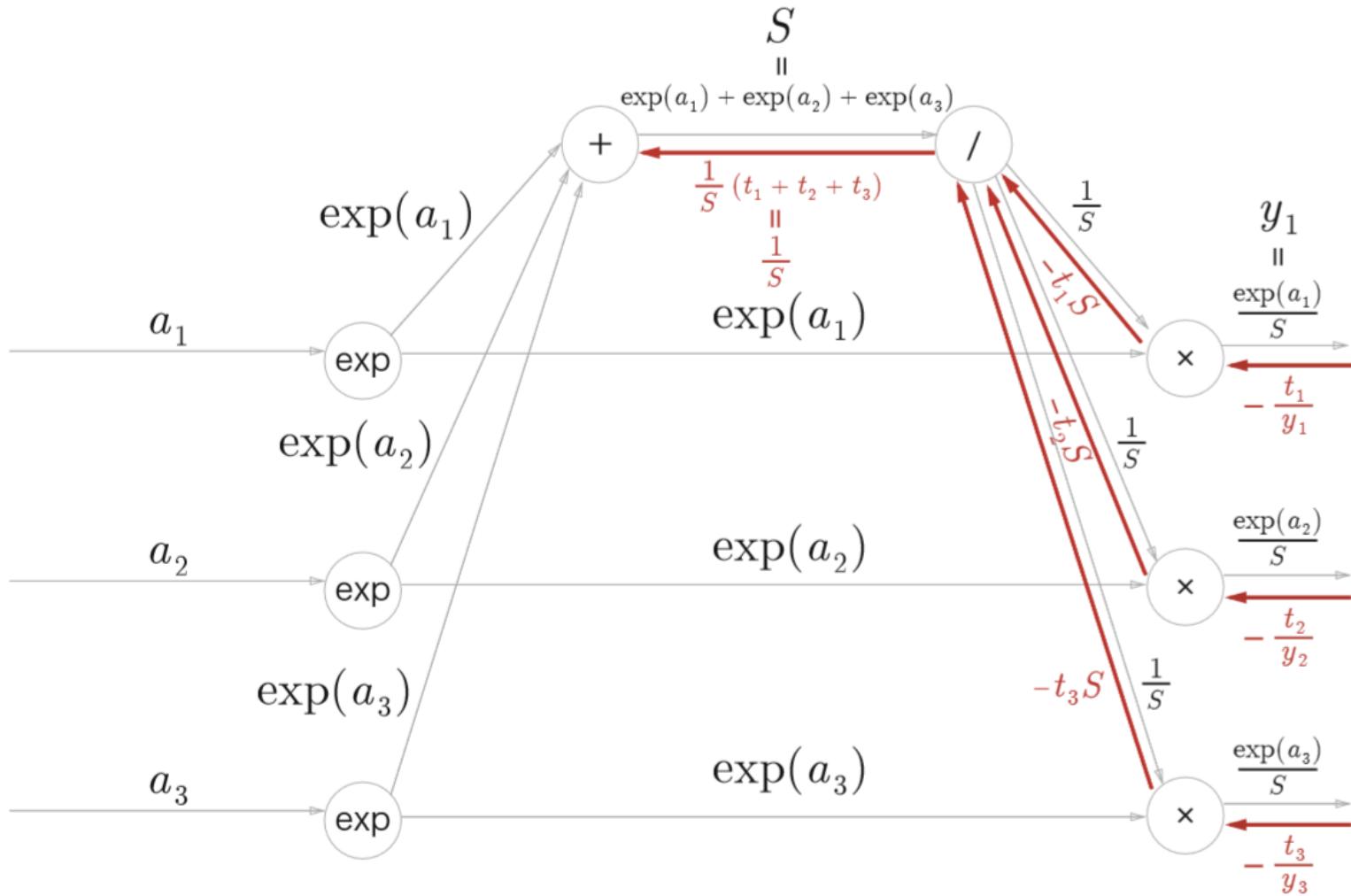
5.6.3 Softmax-with-Loss Layer

Backward propagation



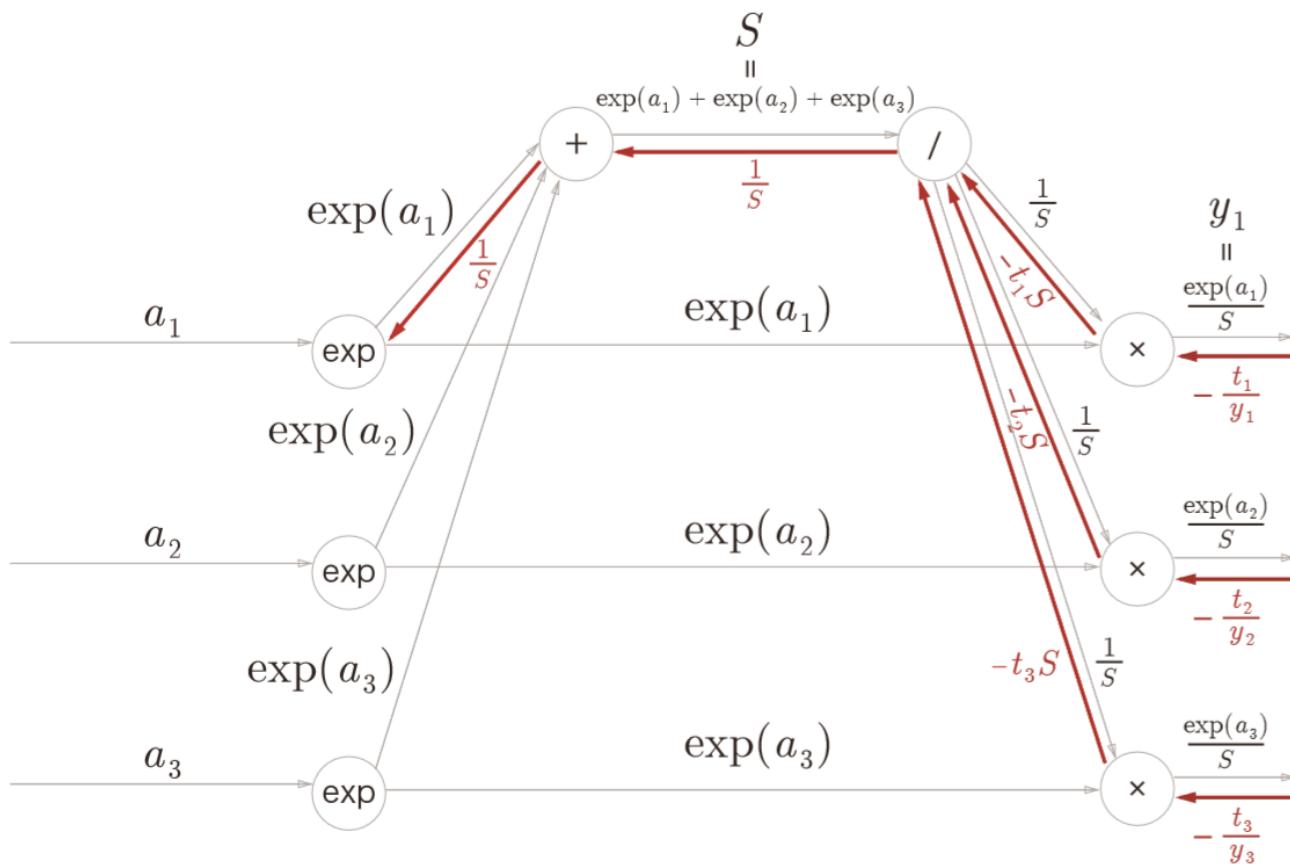
5.6.3 Softmax-with-Loss Layer

Backward propagation



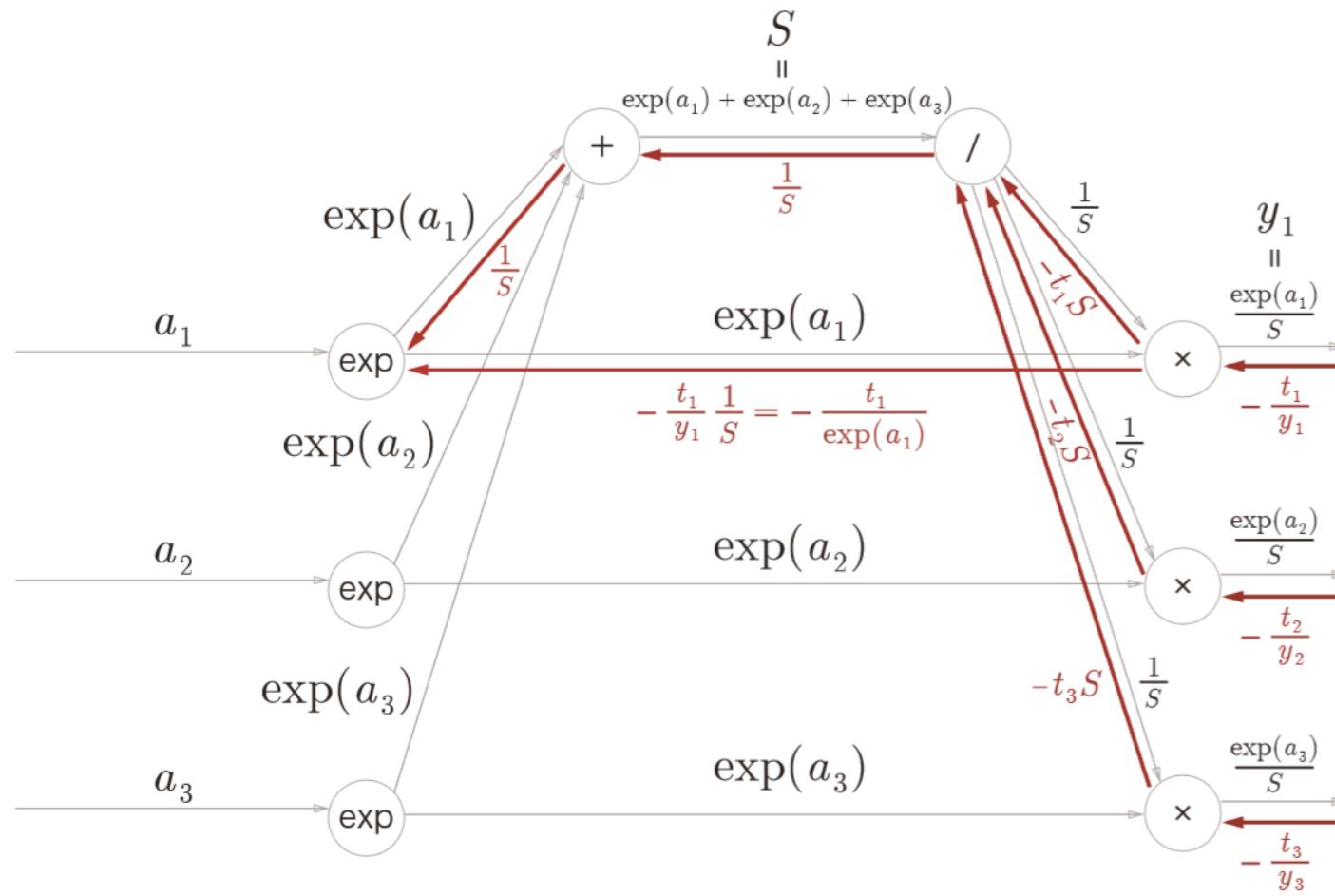
5.6.3 Softmax-with-Loss Layer

Backward propagation



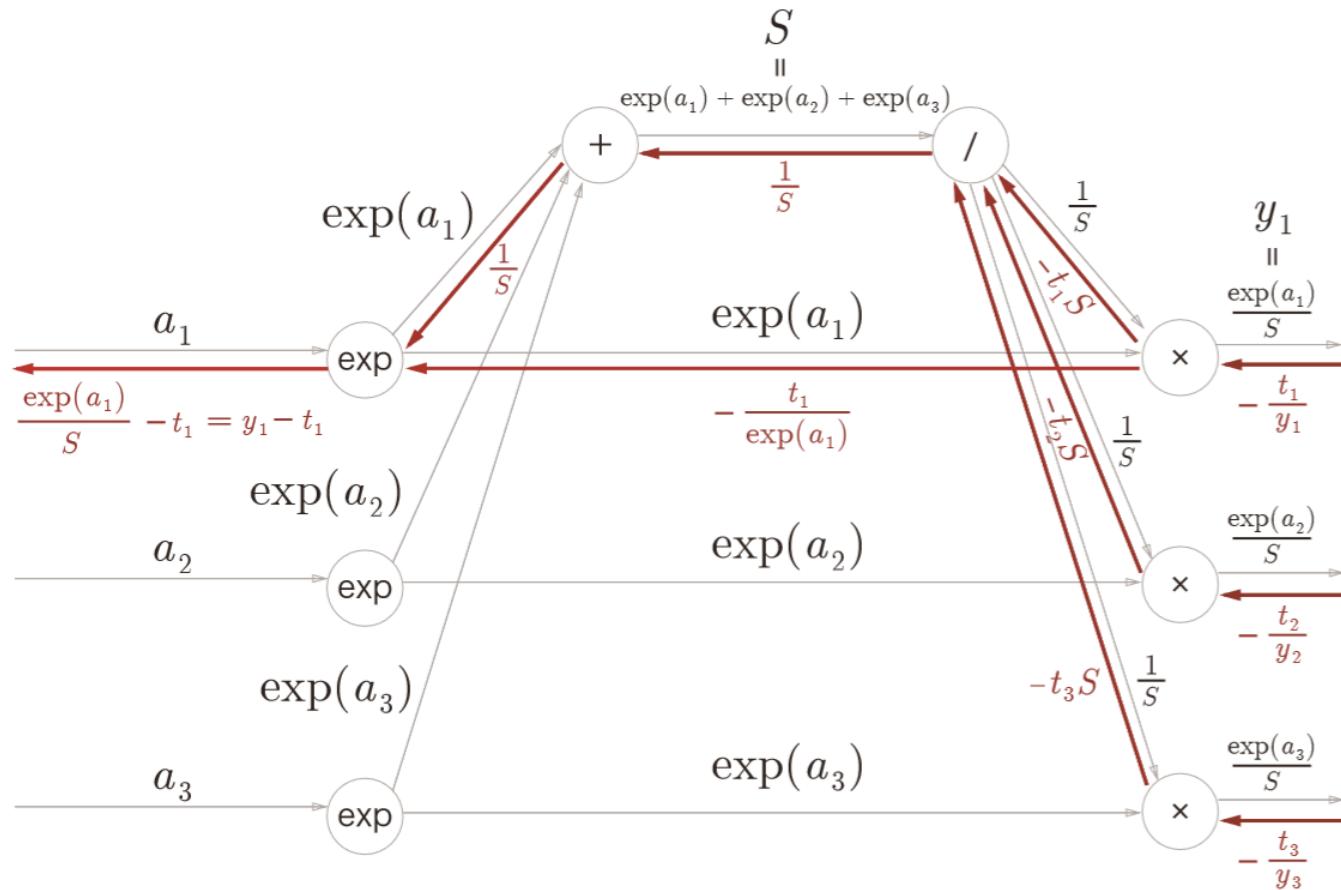
5.6.3 Softmax-with-Loss Layer

Backward propagation



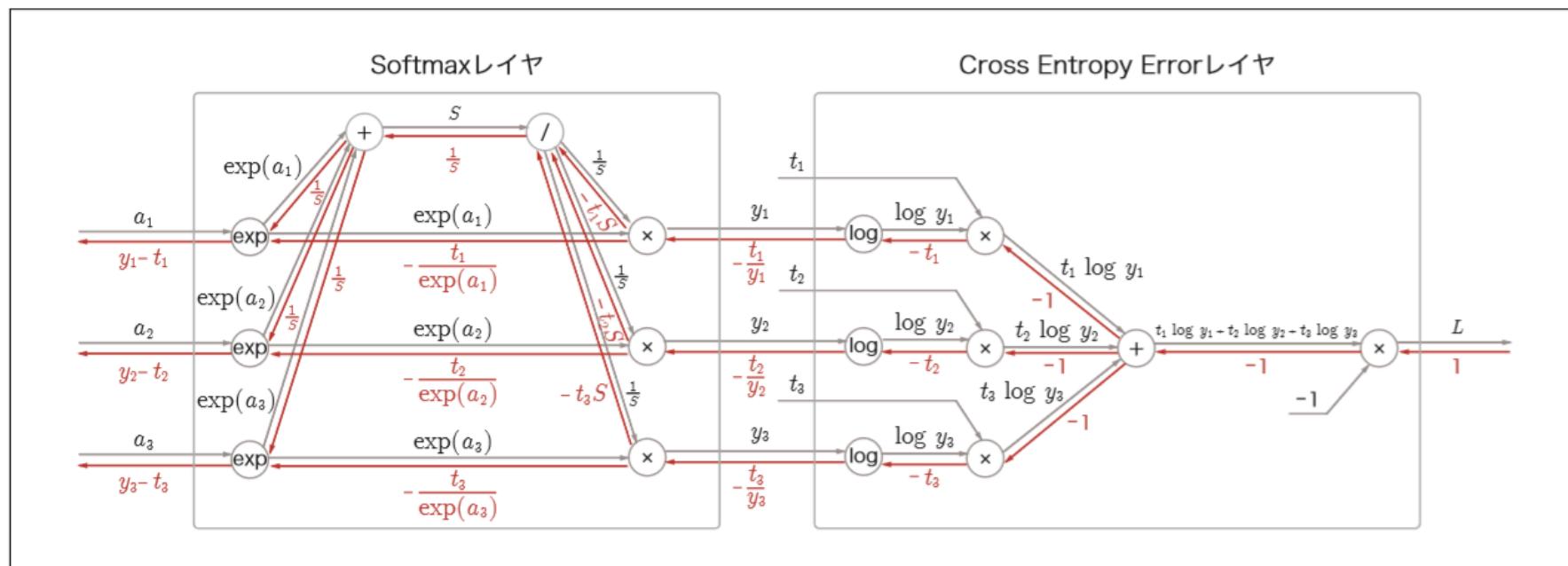
5.6.3 Softmax-with-Loss Layer

Backward propagation



5.6.3 Softmax-with-Loss Layer

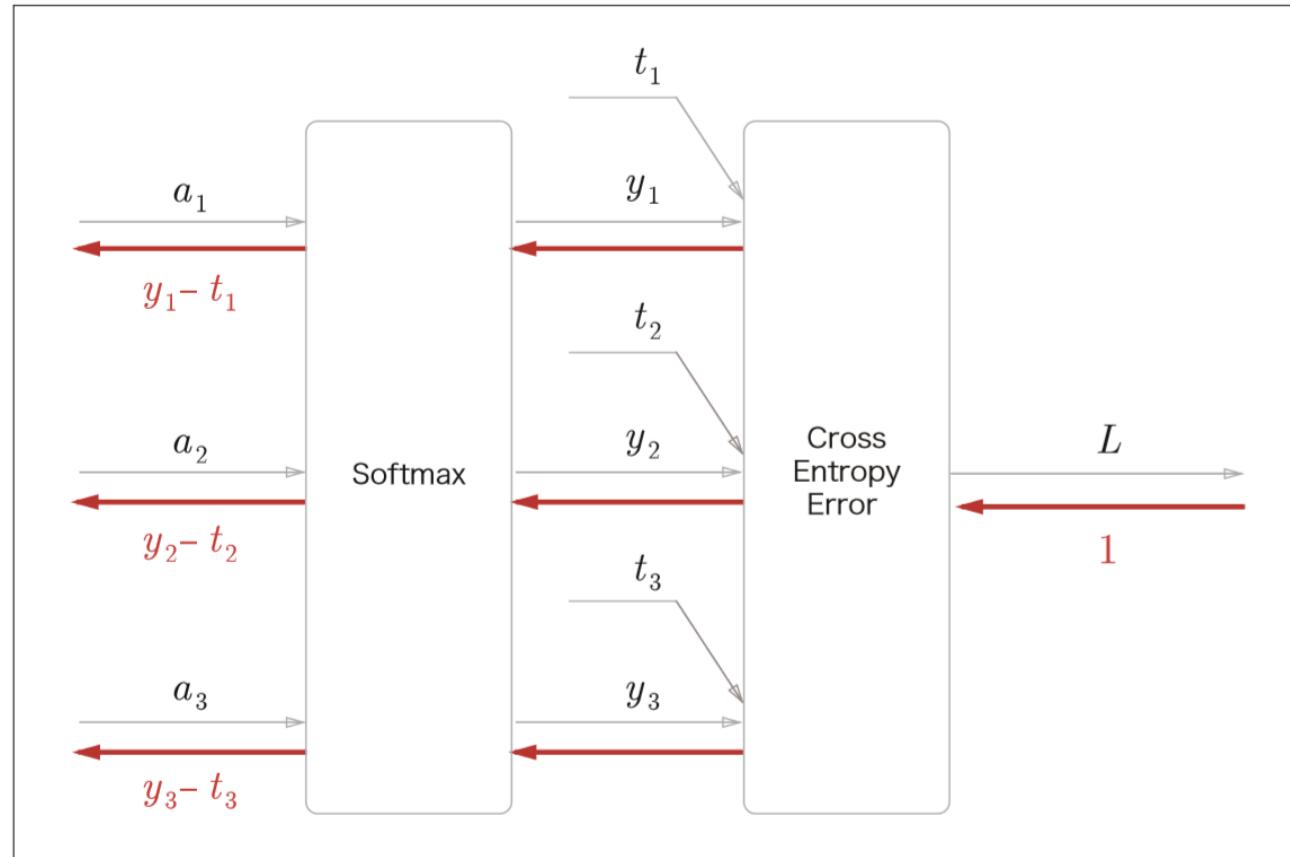
Partial derivative = $y_i - t_i$



5.6.3 Softmax-with-Loss Layer

Simplification

$y_i - t_i =$
(Softmax +
Cross Entropy Error)
Or
(Identity function +
Mean squared error)



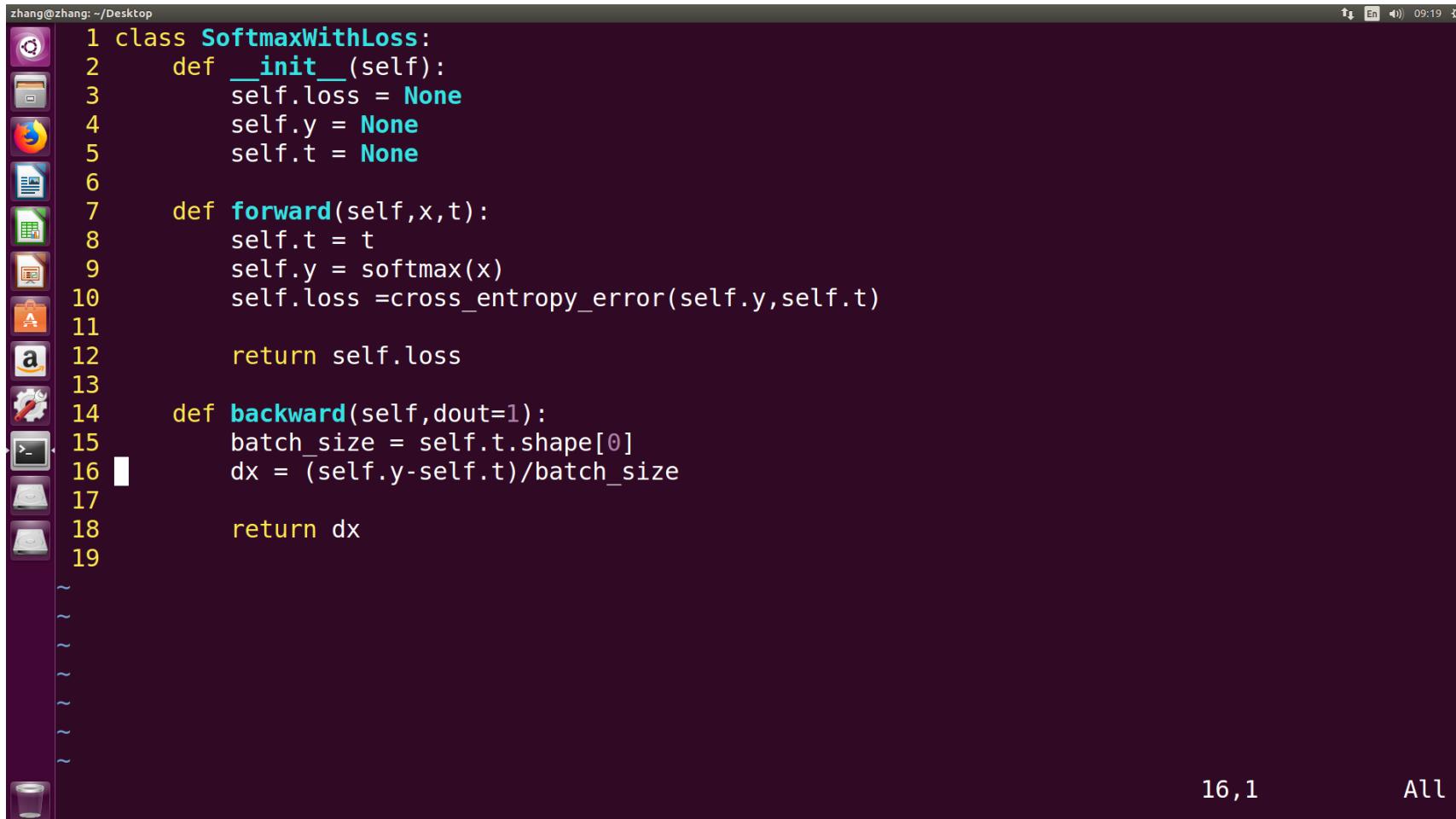
5.6.3 Softmax-with-Loss Layer

As seen above, we can decide how much effort we should take due to the value from the backward propagation.

For example, if the t_{label} is $(0, 1, 0)$, the output value from softmax is $(0.3, 0.2, 0.5)$. Then the value is $(0.3, -0.8, 0.5)$. That means we need a high level learning.

If the output value from softmax is $(0.01, 0.99, 0)$, thus the value is $(0.01, -0.01, 0)$. That means we just need a low level learning.

5.6.3 Softmax-with-Loss Layer



A screenshot of a terminal window on a Linux desktop. The window title is 'zhang@Zhang: ~/Desktop'. The terminal background is dark purple. The code displayed is:

```
1 class SoftmaxWithLoss:
2     def __init__(self):
3         self.loss = None
4         self.y = None
5         self.t = None
6
7     def forward(self,x,t):
8         self.t = t
9         self.y = softmax(x)
10        self.loss = cross_entropy_error(self.y,self.t)
11
12        return self.loss
13
14    def backward(self,dout=1):
15        batch_size = self.t.shape[0]
16        dx = (self.y-self.t)/batch_size
17
18        return dx
19
```

The terminal window also shows a docked application menu with icons for various applications like the Dash, Home, and Dash Help. The bottom right corner of the terminal window displays the text '16,1' and 'All'.

5.7 realization of the back propagation

We can build the neural network based on the description and introduction of layers we just discussed.

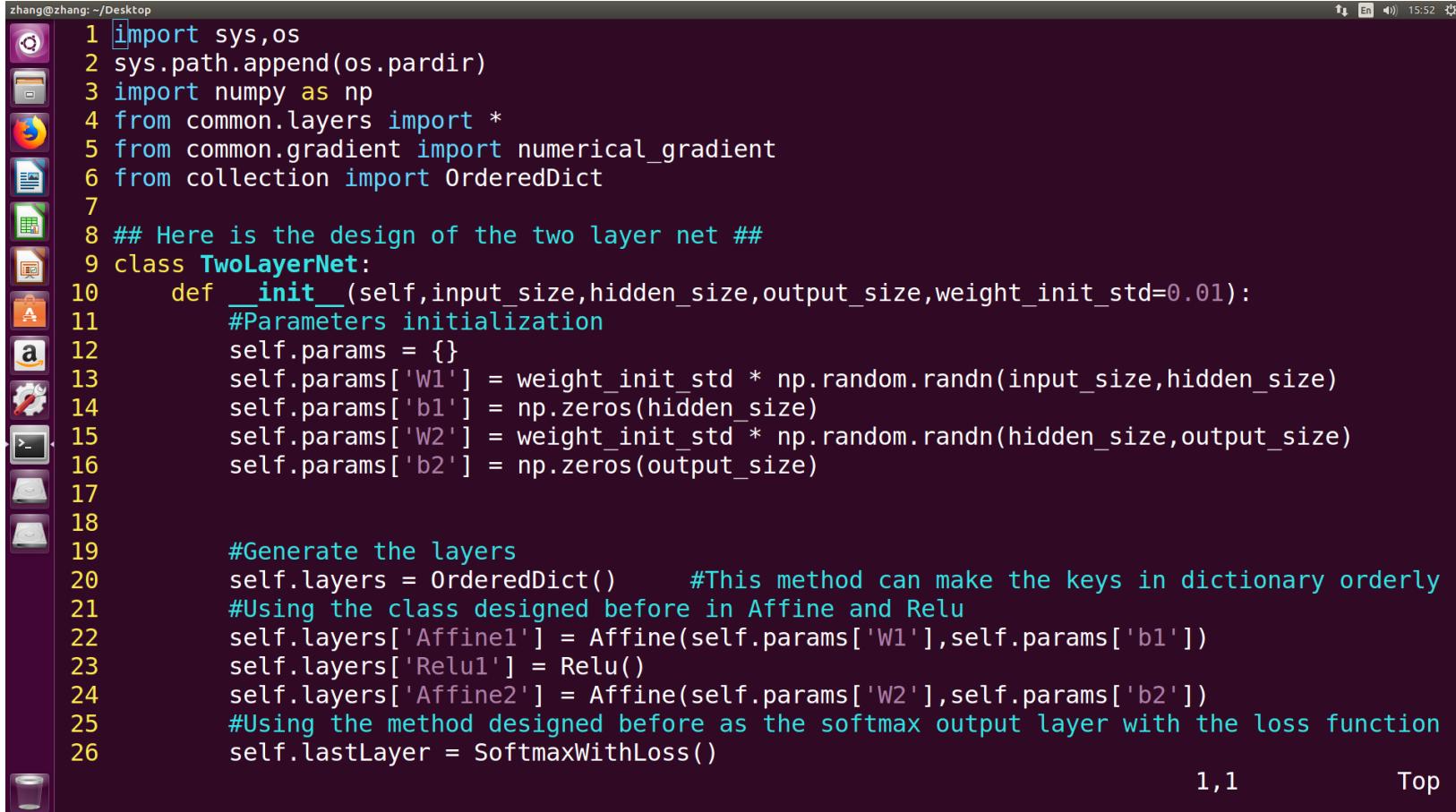
5.7.1 overview of the network's learning

Learning means that we can automatically get the suitable weight and bias from the training data.

Here are the steps:

1. Choose the data randomly from the training data.
2. Calculate the gradient of the loss function.
3. Change the parameters a little along the gradient direction
4. Repeat the 3 steps above

5.7.2 realization of backward propagation network



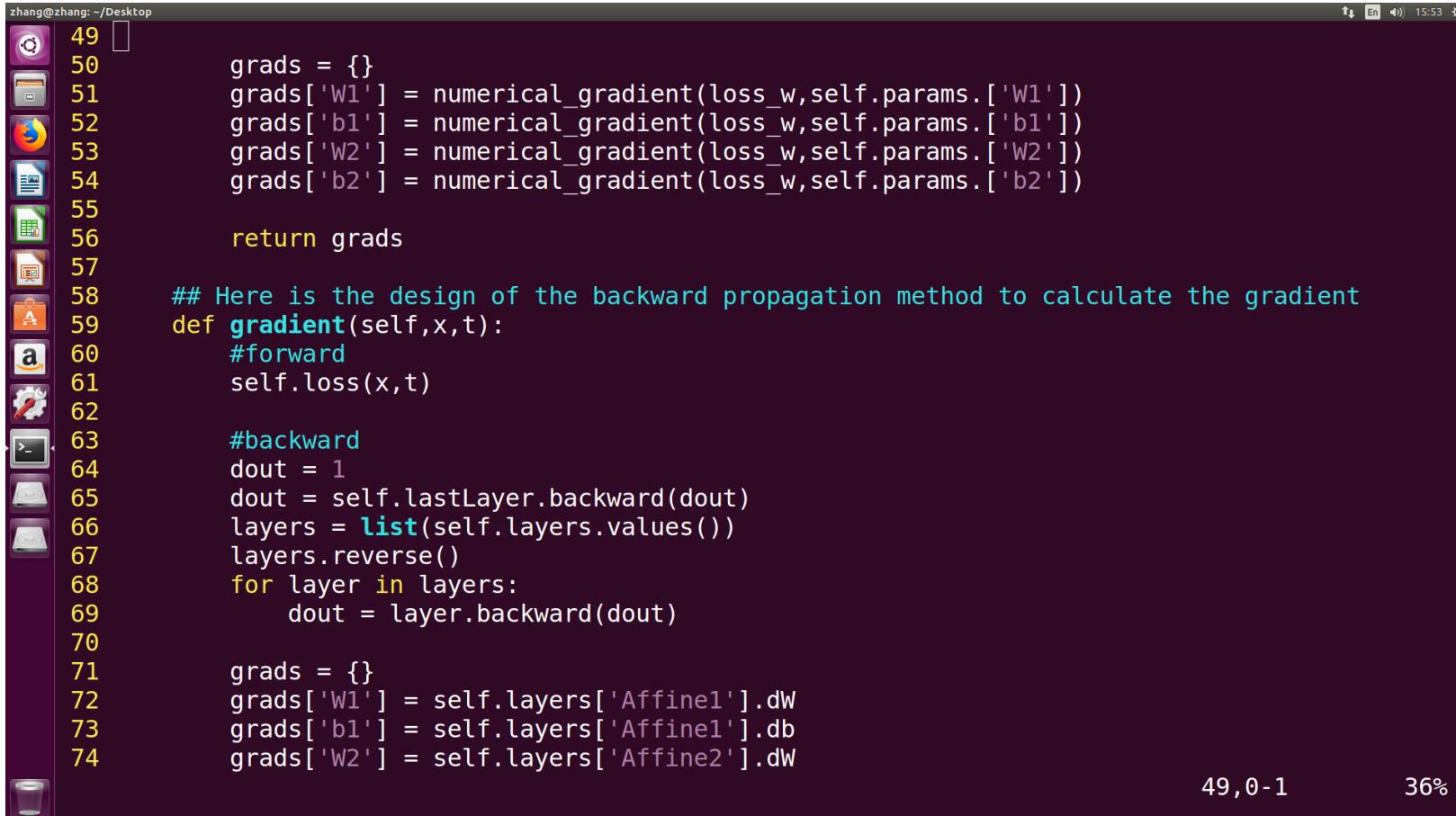
```
zhang@zhang: ~/Desktop
1 import sys,os
2 sys.path.append(os.pardir)
3 import numpy as np
4 from common.layers import *
5 from common.gradient import numerical_gradient
6 from collection import OrderedDict
7
8 ## Here is the design of the two layer net ##
9 class TwoLayerNet:
10     def __init__(self,input_size,hidden_size,output_size,weight_init_std=0.01):
11         #Parameters initialization
12         self.params = {}
13         self.params['W1'] = weight_init_std * np.random.randn(input_size,hidden_size)
14         self.params['b1'] = np.zeros(hidden_size)
15         self.params['W2'] = weight_init_std * np.random.randn(hidden_size,output_size)
16         self.params['b2'] = np.zeros(output_size)
17
18
19         #Generate the layers
20         self.layers = OrderedDict()      #This method can make the keys in dictionary orderly
21         #Using the class designed before in Affine and Relu
22         self.layers['Affinel'] = Affine(self.params['W1'],self.params['b1'])
23         self.layers['Relu1'] = Relu()
24         self.layers['Affine2'] = Affine(self.params['W2'],self.params['b2'])
25         #Using the method designed before as the softmax output layer with the loss function
26         self.lastLayer = SoftmaxWithLoss()
```

1,1 Top

5.7.2 realization of backward propagation network

```
zhang@zhang: ~/Desktop
25     #Using the method designed before as the softmax output layer with the loss function
26     self.lastLayer = SoftmaxWithLoss()
27
28     def predict(self,x):
29         for layer in self.layers.value():
30             x = layer.forward(x)
31
32         return x
33
34     def loss(self,x,t):
35         y = self.predict(x)
36         return self.lastLayer.forward(y,t)
37
38     def accuracy(self,x,t):
39         y = self.predict(x)
40         y = np.argmax(y, axis=1)
41         if t.ndim != 1 : t = np.argmax(t, axis=1)
42
43         accuracy = np.sum(y == t)/float(x.shape[0])
44         return accuracy
45
46     ## Here is the method in chapter 4
47     def numerical_gradient(self,x,t):
48         loss_w = lambda W: self.loss(x,t)
49
50         grads = {}
```

5.7.2 realization of backward propagation network

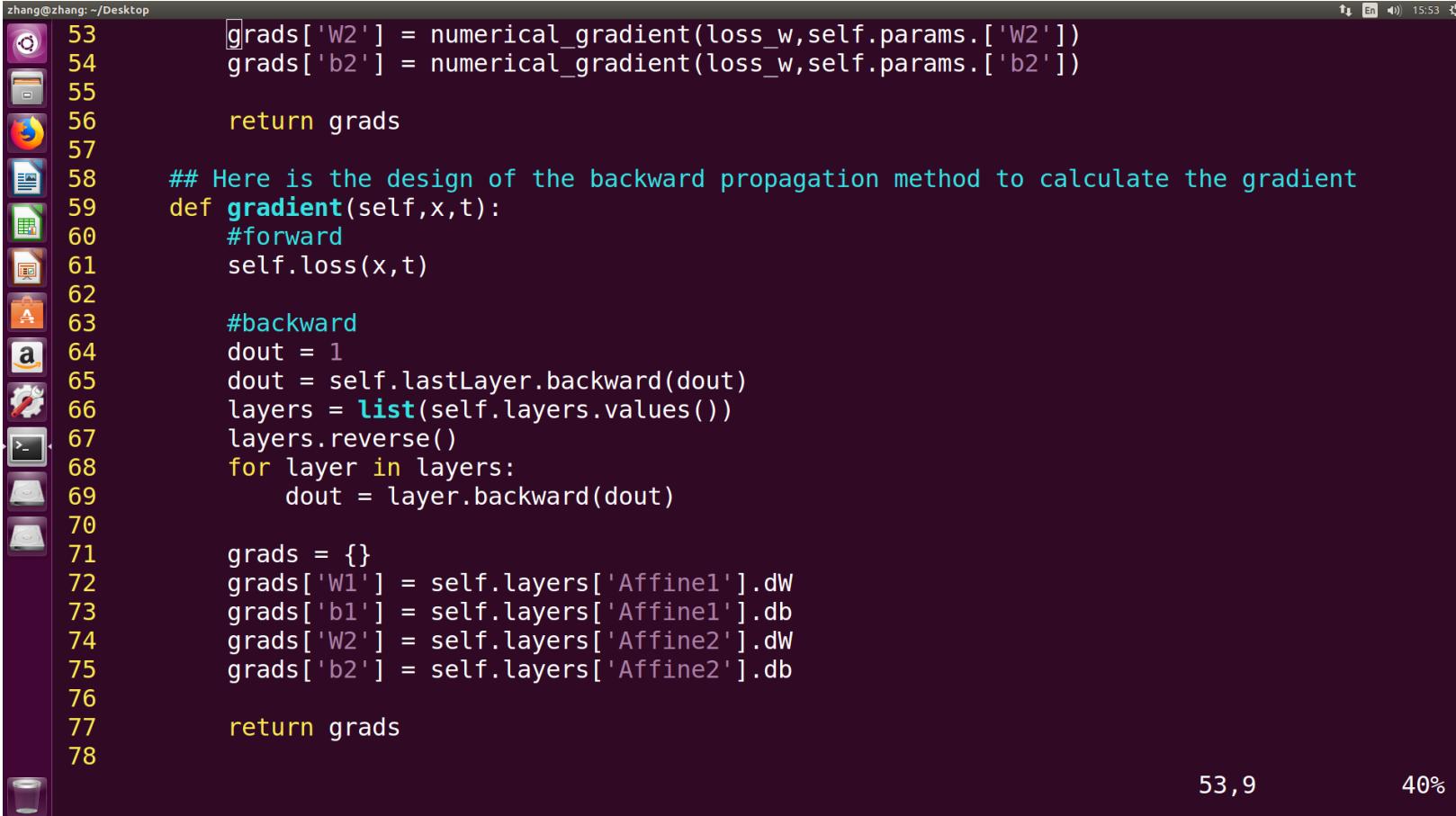


The screenshot shows a terminal window on a Linux desktop environment. The window title is "zhang@zhang: ~/Desktop". The code displayed is a Python script for implementing backward propagation in a neural network. The script defines a class with methods for forward pass, backward pass, and calculating gradients. It uses numerical gradient checking for parameters W1, b1, W2, and b2. The code is color-coded with syntax highlighting.

```
zhang@zhang: ~/Desktop
49
50     grads = {}
51     grads['W1'] = numerical_gradient(loss_w, self.params['W1'])
52     grads['b1'] = numerical_gradient(loss_w, self.params['b1'])
53     grads['W2'] = numerical_gradient(loss_w, self.params['W2'])
54     grads['b2'] = numerical_gradient(loss_w, self.params['b2'])
55
56     return grads
57
58     ## Here is the design of the backward propagation method to calculate the gradient
59     def gradient(self, x, t):
60         #forward
61         self.loss(x, t)
62
63         #backward
64         dout = 1
65         dout = self.lastLayer.backward(dout)
66         layers = list(self.layers.values())
67         layers.reverse()
68         for layer in layers:
69             dout = layer.backward(dout)
70
71         grads = {}
72         grads['W1'] = self.layers['Affine1'].dW
73         grads['b1'] = self.layers['Affine1'].db
74         grads['W2'] = self.layers['Affine2'].dW
```

49, 0-1 36%

5.7.2 realization of backward propagation network



```
zhang@zhang: ~/Desktop
53     grads['W2'] = numerical_gradient(loss_w, self.params()['W2'])
54     grads['b2'] = numerical_gradient(loss_w, self.params()['b2'])
55
56     return grads
57
58     ## Here is the design of the backward propagation method to calculate the gradient
59     def gradient(self, x, t):
60         #forward
61         self.loss(x, t)
62
63         #backward
64         dout = 1
65         dout = self.lastLayer.backward(dout)
66         layers = list(self.layers.values())
67         layers.reverse()
68         for layer in layers:
69             dout = layer.backward(dout)
70
71         grads = {}
72         grads['W1'] = self.layers['Affine1'].dW
73         grads['b1'] = self.layers['Affine1'].db
74         grads['W2'] = self.layers['Affine2'].dW
75         grads['b2'] = self.layers['Affine2'].db
76
77     return grads
78
```

5.7.3 Gradient Calculation and verification

```
zhang@zhang: ~/Desktop
82
83
84 import sys,os
85 sys.path.append(os.pardir)
86 import numpy as np
87 from dataset.mnist import load_mnist
88 from two_layer_net import TwoLayerNet
89
90 #Data reading
91 (x_train,t_train),(x_test,t_test) = load_mnist(normalize=True,one_hot=True)
92
93 network = TwoLayerNet(input_size=784,hidden_size=50,output_size=10)
94
95 x_batch = x_train[:3]
96 t_batch = t_train[:3]
97
98 ## Here is to get the gradient through two different ways ##
99 grad_numerical = network.numerical_gradient(x_batch,t_batch)
100 grad_backprop = network.gradient(x_batch,t_batch)
101
102 ## Here is to compare the difference between the two methods
103 for key in grad_numerical.keys():
104     diff = np.average(np.abs(grad_backprop[key]-grad_numerical[key]))
105     print(key + ":" + str(diff))
106
107
```

82,0-1 62%

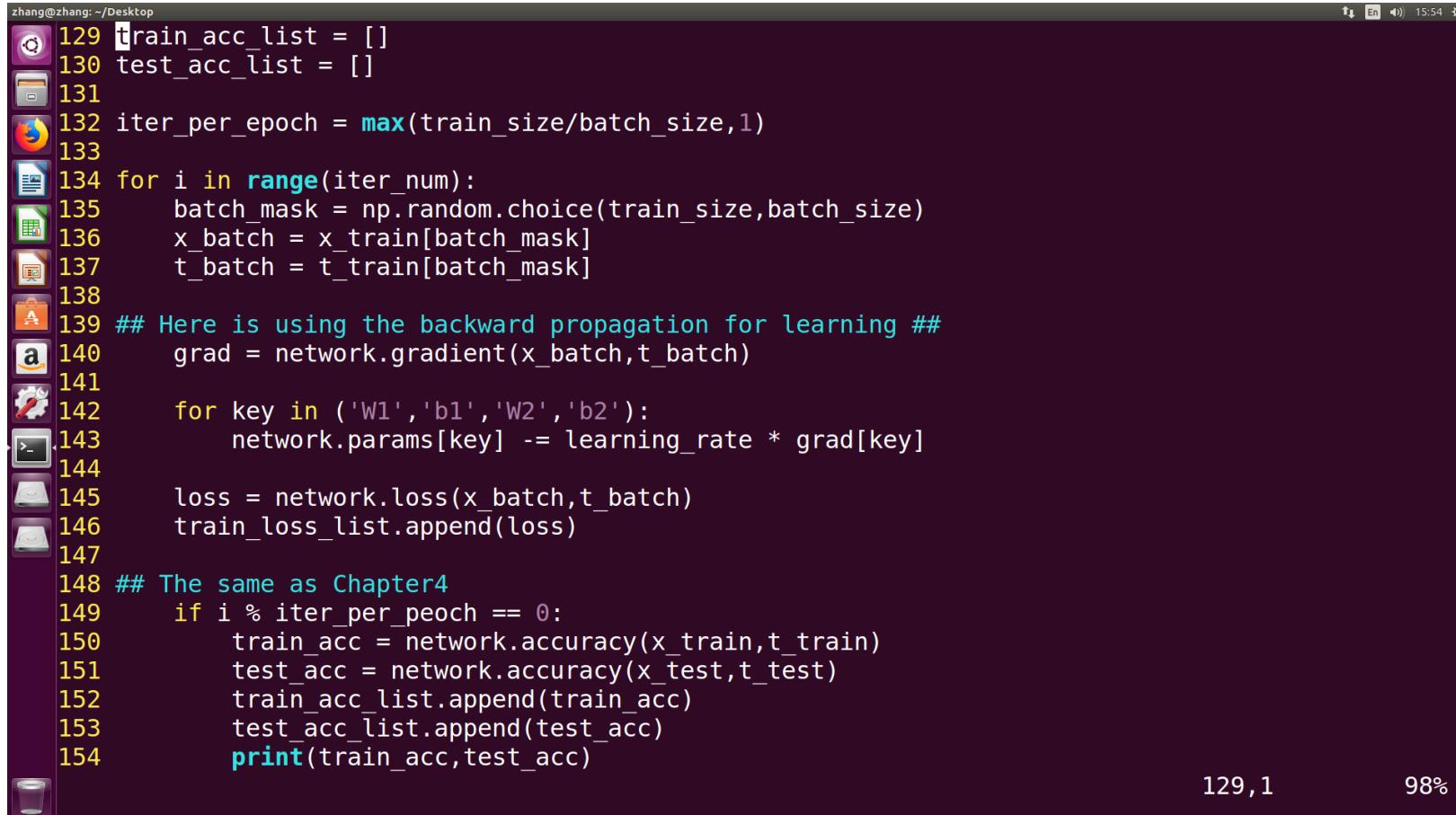
b1:9.70418809871e-13
W2:8.41139039497e-13
b2:1.1945999745e-10
W1:2.2232446644e-13

5.7.4 Learning by backward propagation network

```
zhang@zhang: ~/Desktop
112 import sys,os
113 sys.path.append(os.pardir)
114 import numpy as np
115 from dataset.mnist import load_mnist
116 from two_layer_net import TwoLayerNet
117
118 #Data reading
119 (x_train,t_train),(x_test,t_test) = load_mnist(normalize=True,one_hot=True)
120
121 network = TwoLayerNet(input_size=784,hidden_size=50,output_size=10)
122
123 iters_num = 10000
124 train_size = x_train.shape[0]
125 batch_size = 100
126 learning_rate = 0.1
127
128 train_loss_list = []
129 train_acc_list = []
130 test_acc_list = []
131
132 iter_per_epoch = max(train_size/batch_size,1)
133
134 for i in range(iters_num):
135     batch_mask = np.random.choice(train_size,batch_size)
136     x_batch = x_train[batch_mask]
137     t_batch = t_train[batch_mask]
```

112,1 85%

5.7.4 Learning by backward propagation network



The screenshot shows a terminal window on a dark-themed desktop environment. The terminal window has a title bar with the text "zhang@zhang: ~/Desktop". The window contains a block of Python code. The code is numbered from 129 to 154. It initializes lists for training and testing accuracy, calculates the number of iterations per epoch, and then enters a loop for each epoch. Inside the loop, it samples a batch from the training data, calculates gradients, updates the network's parameters, and computes the loss. After each epoch, it prints the training and testing accuracy. The desktop background is visible behind the terminal window, showing icons for various applications like a browser, file manager, and terminal.

```
zhang@zhang: ~/Desktop
129 train_acc_list = []
130 test_acc_list = []
131
132 iter_per_epoch = max(train_size/batch_size,1)
133
134 for i in range(iter_num):
135     batch_mask = np.random.choice(train_size,batch_size)
136     x_batch = x_train[batch_mask]
137     t_batch = t_train[batch_mask]
138
139 ## Here is using the backward propagation for learning ##
140     grad = network.gradient(x_batch,t_batch)
141
142     for key in ('W1','b1','W2','b2'):
143         network.params[key] -= learning_rate * grad[key]
144
145     loss = network.loss(x_batch,t_batch)
146     train_loss_list.append(loss)
147
148 ## The same as Chapter4
149     if i % iter_per_epoch == 0:
150         train_acc = network.accuracy(x_train,t_train)
151         test_acc = network.accuracy(x_test,t_test)
152         train_acc_list.append(train_acc)
153         test_acc_list.append(test_acc)
154         print(train_acc,test_acc)

129,1      98%
```

5.8 Summary

- 1.Know about the calculation process from the computational graph
- 2.Design the global calculation through the local calculation
- 3.Know about the calculation result from the forward propagation and get the differential result based on the chain rule.
- 4.Design the neural network based on the layers and use the backward propagation method to calculate the gradient.
- 5.Compare the method between the numerical gradient and the backward propagation to confirm the accuracy of the backward propagation

Thanks for your attention