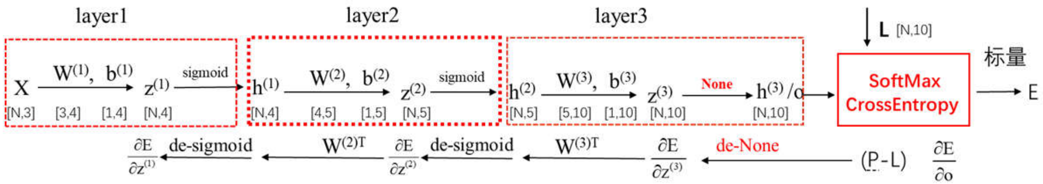
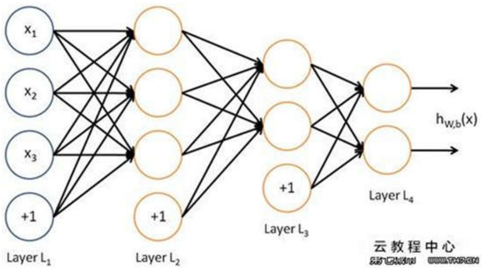
Python编程与人工智能实践

算法篇：神经网络与BP算法

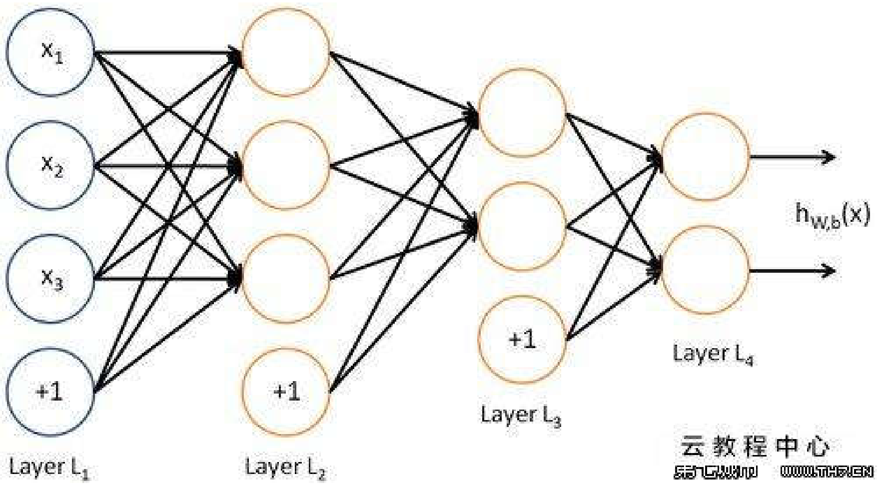


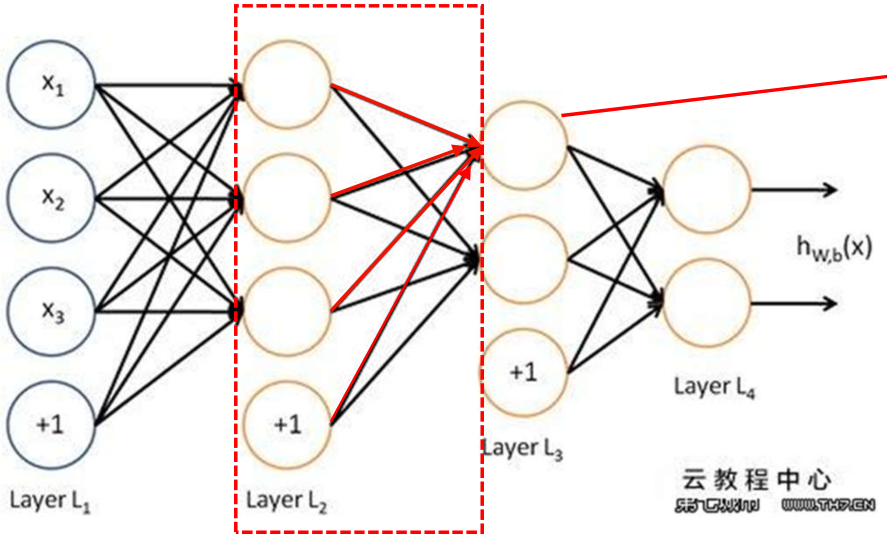


于泓
鲁东大学
信息与电气工程学院
2021.4.4

人工神经网络（Artificial Neural Network）

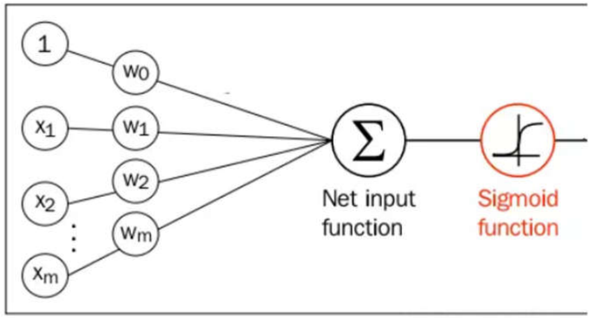
- 人工神经网络（Artificial Neural Networks，简写为ANNs）也简称为神经网络（NNs）或称作连接模型（Connection Model），它是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的。





其中每一个节点类似一个逻辑回归节点

可以认为，一系列的逻辑回归节点，
构成了一个ANN网络



输入层　　隐层　　隐层　　输出层

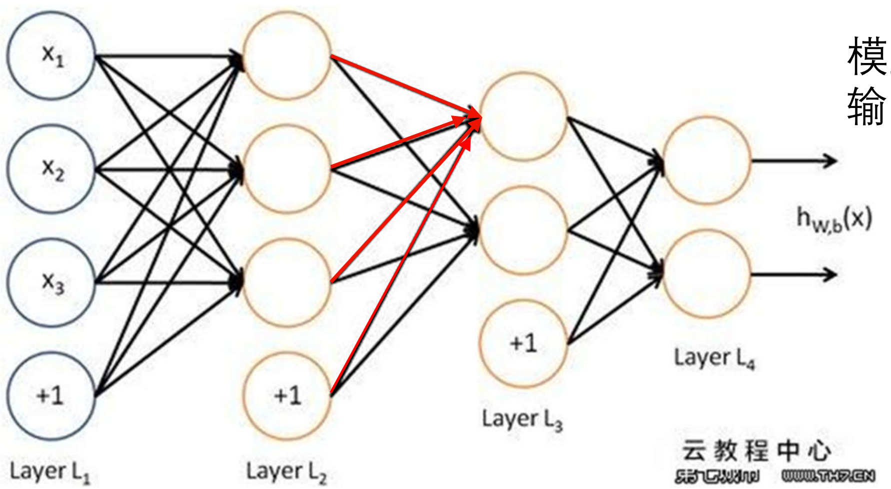
对于一个中间层*i*而言，其输入/输出数学表达式可以写为

$$\mathbf{h}_{out}^{(i)} = f_{actv}^{(i)}(\mathbf{h}_{in}^{(i)}\mathbf{w}^{(i)} + \mathbf{b}^{(i)})$$

上标(*i*)表示第*i*层，
输入$\mathbf{h}_{in}$的维度为 [N ,D_{in}]
输出$\mathbf{h}_{out}$的维度为[N,D_{out}]

需要更新的参数
w 的维度 [D_{in},D_{out}]
b 的维度[1,D_{out}]

$f_{actv}^{(i)}$为激活函数，例如：sigmoid( )



输入层　　隐层　　隐层　　输出层

标签

L

模型输出

o

损失函数

E

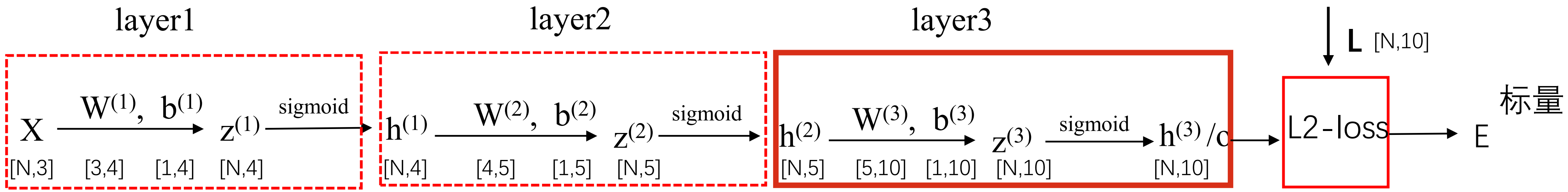
训练模型使E最小

在多分类任务（3类）中

One-hot标签：0 -→ [1,0,0]
1 -→ [0,10]
2 -→ [0,0,1]

最小均方误差损失函数 L2-loss

$$E=\sum_{i=1}^{N}\left\|\mathbf{L}_i-\mathbf{o}_i\right\|_2$$



$$\frac{\partial E}{\partial z^{(3)}} \xleftarrow{\text{de-sigmoid}} (o-L)$$

根据链式求导

输入的转置　　误差

$$\overset{[5,10]}{\frac{\partial E}{\partial W^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial W^{(3)}} = \overset{[5,N]}{h^{(2)T}}\ \overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

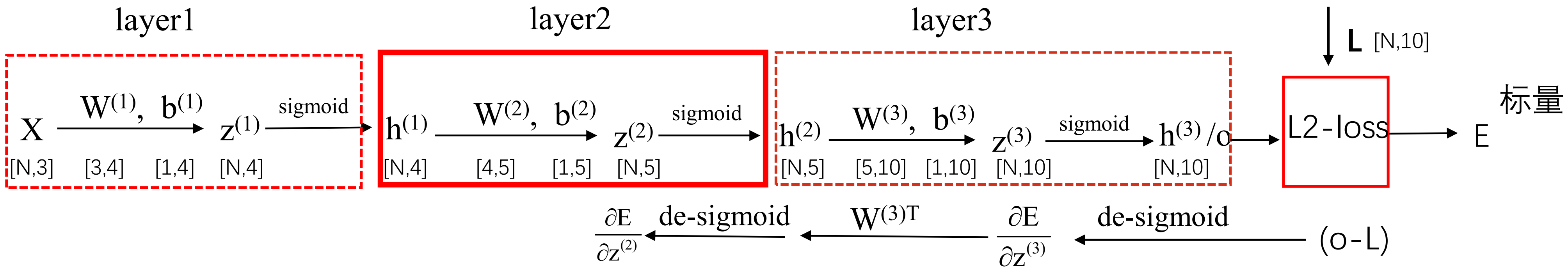
$$\overset{[1,10]}{\frac{\partial E}{\partial b^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial b^{(3)}} = \overset{[1,N]}{\underset{\text{全1}}{I^{T}}}\ \overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

$$\frac{\partial E}{\partial z^{(3)}} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z^{(3)}} = \overset{[N,10]}{(o-L)} \underset{\text{逐元素乘}}{*} \overset{[N,10]}{(h^{(3)})} * \overset{[N,10]}{(1-h^{(3)})}$$

$$E = \frac{1}{2}(o-L)^2, \frac{\partial E}{\partial o} = o - L$$

$$h = \frac{1}{1+e^{-z}}, h' = -\frac{-e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = h(1-h)$$

领用梯度下降
进行参数$w^{(i)}$,$b^{(i)}$的更新

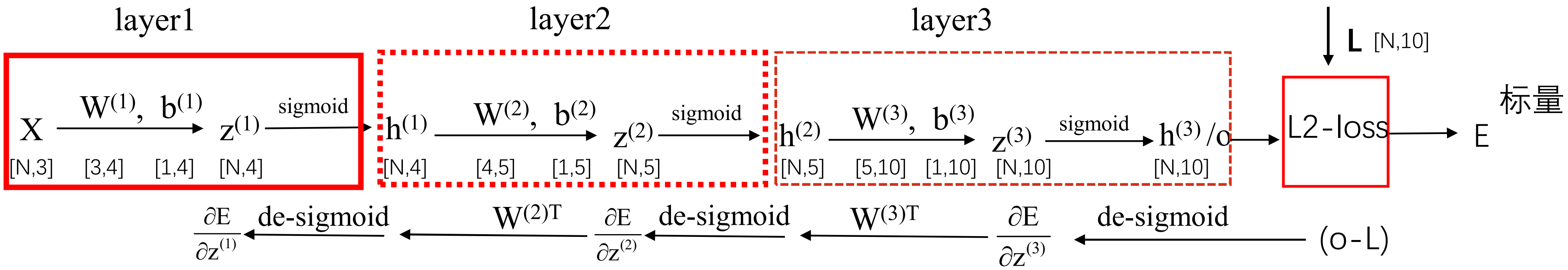


根据链式求导

$$\overset{[4,5]}{\frac{\partial E}{\partial W^{(2)}}}=\frac{\partial E}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial W^{(2)}}=\overset{[4,N]}{h^{(1)T}}\overset{[N,5]}{\frac{\partial E}{\partial z^{(2)}}}$$

$$\overset{[1,5]}{\frac{\partial E}{\partial b^{(2)}}}=\frac{\partial E}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial b^{(2)}}=\overset{[1,N]}{I^{T}}\overset{[N,5]}{\frac{\partial E}{\partial z^{(2)}}}$$

$$\overset{[N,5]}{\frac{\partial E}{\partial z^{(2)}}}=\frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial z^{(2)}}=\overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}\overset{[10,5]}{W^{(3)T}}\overset{[N,5]}{*(h^{(2)})*(1\text{-}h^{(2)})}$$

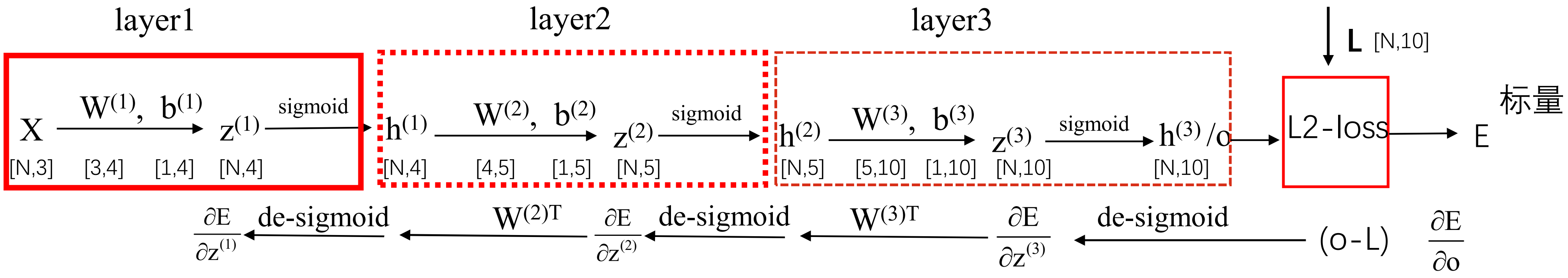


根据链式求导

$$\overset{[3,4]}{\frac{\partial E}{\partial W^{(1)}}} = \frac{\partial E}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial W^{(1)}} = \overset{[3,N]}{X^{T}}\overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}}$$

$$\overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}} = \frac{\partial E}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial h^{(1)}}\frac{\partial h^{(1)}}{\partial z^{(1)}} = \overset{[N,5]}{\frac{\partial E}{\partial z^{(2)}}}\overset{[5,4]}{W^{(2)T}} * \overset{[N,4]}{(h^{(1)})*(1\text{-}h^{(1)})}$$

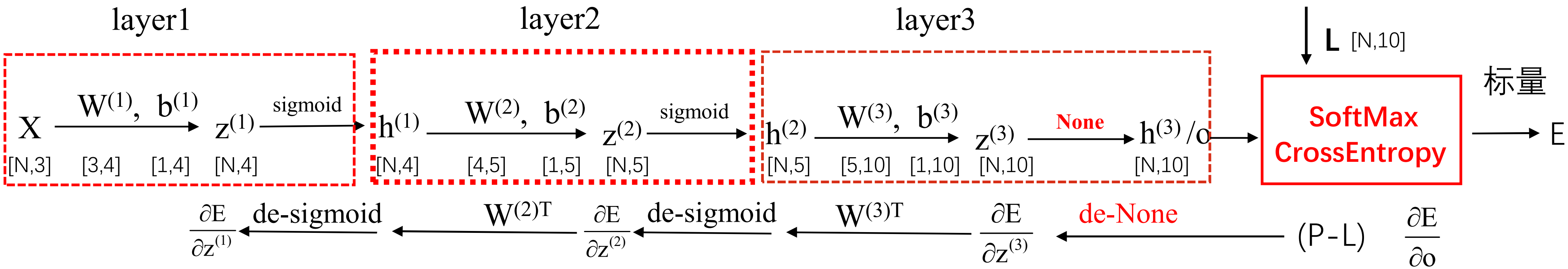
$$\overset{[1,4]}{\frac{\partial E}{\partial b^{(1)}}} = \frac{\partial E}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial b^{(1)}} = \overset{[1,N]}{I^{T}}\overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}}$$



参数的导数 = 输入的转置 x 误差

误差：损失函数对网络输出的导数 ，$\frac{\partial E}{\partial o}$ 由后往前逐级传递得到

Back Propagation（BP算法）
误差反向传播



SoftMax交叉熵是分类任务中更常使用的损失函数

以三分类任务为例子

$$CE(o,L)=-l_1\log(P_1)-l_2\log(P_2)-l_3\log(P_3)$$

$$P_i=\frac{e^{o_i}}{e^{o_1}+e^{o_2}+e^{o_3}}$$

因为L为one-hot标签所以只有l_k非0

$$CE=-\log(P_k)=\log(e^{o_1}+e^{o_2}+e^{o_3})-o_k$$

$$\frac{\partial CE}{\partial o_i}=\begin{cases}\dfrac{e^{o_i}}{e^{o_1}+e^{o_2}+e^{o_3}}-1=P_i-1 & i=k\\[2ex] \dfrac{e^{o_i}}{e^{o_1}+e^{o_2}+e^{o_3}}-0=P_i-0 & i\text{不等于}k\end{cases}=P-L$$

Sigmoid 前向 反向

```python
import numpy as np

def sigmod(z):
    h = 1./(1+np.exp(-z))
    return h


def de_sigmoid(h):
    return h*(1-h)
```

无激活函数 前向，反向

```python
def no_active(z):
    h = z
    return h

def de_no_active(h):
    return np.ones(h.shape)
```

L2 损失函数 前向 反向

```python
# o Nxc
# lab Nxc
def loss_L2(o,lab):
    diff = lab-o
    sqrDiff = diff ** 2
    return 0.5*np.sum(sqrDiff)

def de_loss_L2(o,lab):
    return o-lab
```

softmax交叉熵损失函数 前向 反向

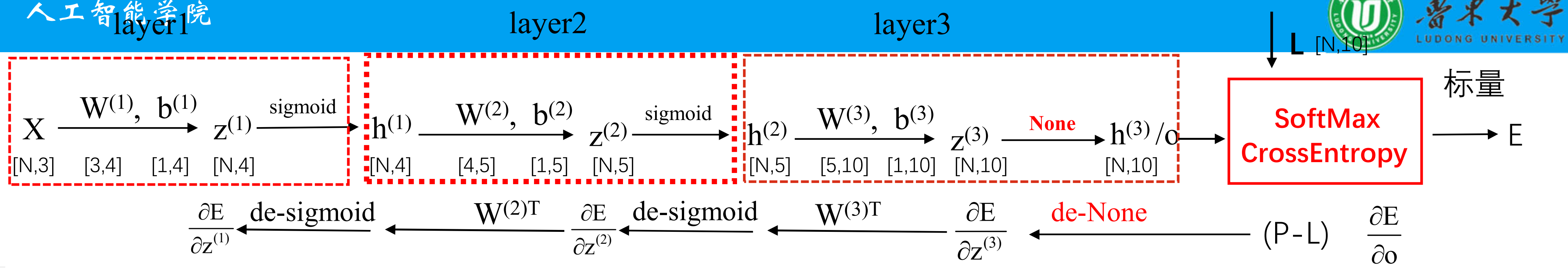
```python
def loss_CE(o,lab):
    p = np.exp(o)/np.sum(np.exp(o),axis=1,keepdims=True)
    loss_ce = np.sum(-lab*np.log(p))
    return loss_ce

def de_loss_CE(o,lab):
    p = np.exp(o)/np.sum(np.exp(o),axis=1,keepdims=True)
    return p-lab
```

构建网络，对权重w 偏置b 进行初始化

```python
# dim_in:输入特征的维度
# list_num_hidden:  每层输出节点的数目
# list_act_funs:  每层的激活函数
# list_de_act_funs: 反向传播时的函数

def bulid_net(dim_in,list_num_hidden,
              list_act_funs,list_de_act_funs):
    layers=[]

    # 逐层的进行网络构建
    for i in range(len(list_num_hidden)):
        layer = {}

        # 定义每一层的权重
        if i ==0:
            layer["w"]= 0.2*np.random.randn(dim_in,list_num_hidden[i])-0.1
        else:
            layer["w"]= 0.2*np.random.randn(list_num_hidden[i-1],list_num_hidden[i])-0.1

        # 定义每一层的偏置
        layer["b"] = 0.1*np.ones([1,list_num_hidden[i]])
        layer["act_fun"]= list_act_funs[i]
        layer["de_act_fun"]= list_de_act_funs[i]
        layers.append(layer)

    return layers
```

```python
# 返回每一层的输入
# 与最后一层的输出
def fead_forward(datas,layers):
    input_layers = []

    for i in range(len(layers)):
        layer = layers[i]
        if i ==0:
            inputs = datas
            z = np.dot(inputs,layer["w"]) + layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
        else:
            inputs = h
            z = np.dot(inputs,layer["w"])+ layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
    return input_layers,h
```

前向传播记录每一层的输入
以及最后一层的输出即o




```
# 进行参数更新更新
def updata_wb(datas,labs,layers, loss_fun,de_loss_fun,alpha=0.01):
    N,D = np.shape(datas)
    # 进行前馈操作
    inputs,output = fead_forward(datas,layers)

    # 计算 loss
    loss = loss_fun(output,labs)

    #从后向前计算
    deltas0 = de_loss_fun(output,labs)

    # 从后向前计算误差
    deltas =[]
    for i in range(len(layers)):
        index = -i-1
        if i ==0:
            delta = deltas0*layers[index]["de_act_fun"](output)
        else:
            h = inputs[index+1]
            delta = np.dot(delta,layers[index+1]["w"].T)*layers[index]["de_act_fun"](h)

        deltas.insert(0,delta)
```

$$\frac{\partial E}{\partial z^{(3)}} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z^{(3)}} = (o\text{-}L)*(h^{(3)})*(1\text{-}h^{(3)})$$

$$\frac{\partial E}{\partial z^{(2)}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial z^{(2)}} = \frac{\partial E}{\partial z^{(3)}} W^{(3)T}*(h^{(2)})*(1\text{-}h^{(2)})$$

```python
# 利用误差 对每一层的权重进行修成
for i in range(len(layers)):
    # 计算 dw 与 db
    dw = np.dot(inputs[i].T,deltas[i])
    db = np.sum(deltas[i],axis=0,keepdims=True)
    # 梯度下降
    layers[i]["w"] = layers[i]["w"] - alpha*dw
    layers[i]["b"] = layers[i]["b"] - alpha*db

return layers,loss
```

$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}} = h^{(1)T} \frac{\partial E}{\partial z^{(2)}}$$

$$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = I^{T} \frac{\partial E}{\partial z^{(2)}}$$

相当于对第0维求和

测试结果

```python
def test_accuracy(datas,labs_true,layers):
    _,output = fead_forward(datas,layers)
    lab_det = np.argmax(output,axis=1)
    labs_true = np.argmax(labs_true,axis=1)
    N_error = np.where(np.abs(labs_true-lab_det)>0)[0].shape[0]

    error_rate = N_error/np.shape(datas)[0]
    return error_rate
```

在鸢尾花数据集上测试

```
def load_dataset_iris(file_data,N_train):
    # 数据读取
    datas = np.loadtxt(file_data,dtype = np.float, delimiter = ',',usecols=(0,1,2,3))
    labs = np.loadtxt(file_data,dtype = str, delimiter = ',',usecols=(4))
    N,D = np.shape(datas)
    N_test = N-N_train
    unqiue_labs = np.unique(labs).tolist()

    dic_str2index={}
    dic_index2str={}
    for i in range(len(unqiue_labs)):
        lab_str = unqiue_labs[i]
        dic_str2index[lab_str] =i
        dic_index2str[i]=lab_str

    labs_onehot = np.zeros([N,len(unqiue_labs)])
    for i in range(N):
        labs_onehot[i,dic_str2index[labs[i]]]=1

    perm = np.random.permutation(N)
    index_train = perm[:N_train]
    index_test = perm[N_train:]

    data_train = datas[index_train,:]
    lab_train_onehot = labs_onehot[index_train,:]

    data_test = datas[index_test,:]
    lab_test_onehot = labs_onehot[index_test]

    return data_train,lab_train_onehot,data_test,lab_test_onehot,dic_index2str
```

```
if __name__=="__main__":

    file_data = 'iris.data'

    data_train,lab_train_onehot,data_test,lab_test_onehot,dic_index2str =load_dataset_iris(file_data,100)

    N,dim_in = np.shape(data_train)
    # 定义网络结构
    list_num_hidden=[20,20,3]
    list_act_funs =[sigmod,sigmod,no_active]
    list_de_act_funs=[de_sigmoid,de_sigmoid,de_no_active]

    # 定义损失函数
    loss_fun = loss_CE
    de_loss_fun=de_loss_CE

    # loss_fun = loss_L2
    # de_loss_fun=de_loss_L2

    layers = bulid_net(dim_in,list_num_hidden,
          list_act_funs,list_de_act_funs)
```

```python
# 进行训练
n_epoch = 200
batchsize =4
N_batch = N//batchsize
for i in range(n_epoch):
    # 数据打乱
    rand_index  = np.random.permutation(N).tolist()
    # 每个batch 更新一下weight
    loss_sum =0
    for j in range(N_batch):
        index = rand_index[j*batchsize:(j+1)*batchsize]
        batch_datas = data_train[index]
        batch_labs = lab_train_onehot[index]
        layers,loss = updata_wb(batch_datas,batch_labs,layers,loss_fun,de_loss_fun,alpha=0.01)
        loss_sum = loss_sum+loss

    error = test_accuracy(data_train,lab_train_onehot,layers)
    print("epoch %d  error  %.2f%%  loss_all %.2f"%(i,error*100,loss_sum))

#进行测试
error = test_accuracy(data_test,lab_test_onehot,layers)
print(error*100)
```

手写文字数据集 MINIST

包含70000张图片，每张大小28*28=784

```python
import numpy as np
from NN_BP import *

def load_mnist(file_data,file_lab):
    # 加载训练数据
    data = np.load(file_data)
    lab = np.load(file_lab)
    N,D = np.shape(data)

    # 构造 one-hot 标签
    lab_onehot = np.zeros([N,10])
    for i in range(N):
        id = int(lab[i,0])
        lab_onehot[i,id]=1
    data = data.astype(np.float)/255.0
    return data,lab_onehot
```

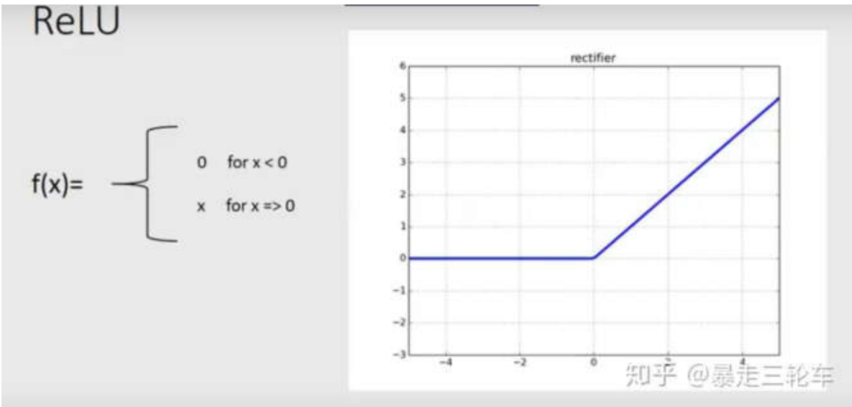
```python
if __name__=="__main__":

    # 加载训练数据
    train_data,train_lab_onehot=load_mnist("train_data.npy","train_lab.npy")
    N,D = np.shape(train_data)

    # 搭建网络
    # 定义网络结构
    list_num_hidden=[30,20,10]

    # list_act_funs =[sigmod,sigmod,sigmod]
    # list_de_act_funs=[de_sigmoid,de_sigmoid,de_sigmoid]
    # # 定义损失函数
    # loss_fun = loss_L2
    # de_loss_fun=de_loss_L2

    list_act_funs =[sigmod,sigmod,no_active]
    list_de_act_funs=[de_sigmoid,de_sigmoid,de_no_active]
    # 定义损失函数
    loss_fun = loss_CE
    de_loss_fun=de_loss_CE

    layers = bulid_net(D,list_num_hidden,
              list_act_funs,list_de_act_funs)
```

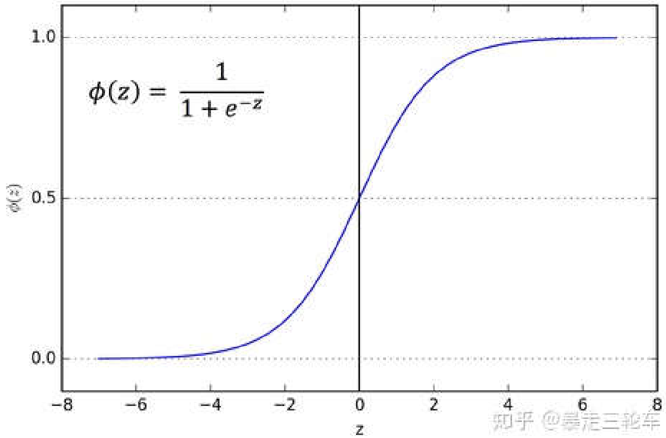
```python
# 进行训练
n_epoch = 50
batchsize =20
N_batch = N//batchsize
for i in range(n_epoch):
    # 数据打乱
    rand_index  = np.random.permutation(N).tolist()
    # 每个batch 更新一下weight
    loss_sum =0
    for j in range(N_batch):
        index = rand_index[j*batchsize:(j+1)*batchsize]
        batch_datas = train_data[index]
        batch_labs = train_lab_onehot[index]
        layers,loss = updata_wb(batch_datas,batch_labs,layers,loss_fun,de_loss_fun,alpha=0.03)
        # print("epoch %d  batch %d  loss %.2f"%(i,j,loss/batchsize))
        loss_sum = loss_sum+loss

    error = test_accuracy(train_data,train_lab_onehot,layers)
    print("epoch %d  error  %.2f%%  loss_all %.2f"%(i,error*100,loss_sum/(N_batch*batchsize)))

np.save("model.npy",layers)

# 加载测试数据
test_data,test_lab_onehot=load_mnist("test_data.npy","test_lab.npy")
layers = np.load("model.npy",allow_pickle=True)

error = test_accuracy(test_data,test_lab_onehot,layers)
print("Accuarcy on Test Data %.2f %%"%((1-error)*100))
```

```
epoch 38  error  1.47%  loss_all 0.05
epoch 39  error  1.12%  loss_all 0.05
epoch 40  error  1.32%  loss_all 0.05
epoch 41  error  1.49%  loss_all 0.05
epoch 42  error  1.20%  loss_all 0.04
epoch 43  error  1.27%  loss_all 0.04
epoch 44  error  0.99%  loss_all 0.04
epoch 45  error  1.26%  loss_all 0.05
epoch 46  error  1.42%  loss_all 0.04
epoch 47  error  1.04%  loss_all 0.04
epoch 48  error  1.52%  loss_all 0.04
epoch 49  error  1.00%  loss_all 0.04
Accuarcy on Test Data 96.17 %
```

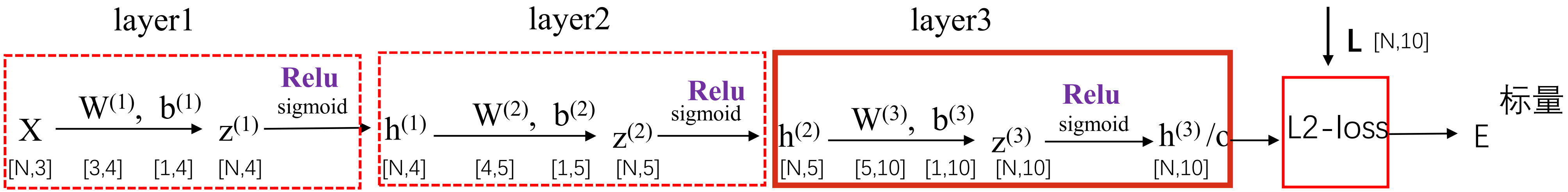
添加 relu 激活函数

线性整流函数 （Rectified Linear Unit, ReLU ），又称 修正线性单元,
是一种 人工神经网络中常用的激活函数（activation function），
通常指代以 斜坡函数 及其变种为代表的 非线性函数 。





与sigmoid函数相比较Relu可以使神经元输出更强的激励信号



$$\frac{\partial E}{\partial z^{(3)}} \xleftarrow{\text{de-sigmoid}} (o\text{-}L)$$

根据链式求导

输入的转置　　误差

$$\overset{[5,10]}{\frac{\partial E}{\partial W^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial W^{(3)}} = \overset{[5,N]}{h^{(2)T}}\ \overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

$$\frac{\partial E}{\partial z^{(3)}} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z^{(3)}} = \overset{[N,10]}{(o\text{-}L)} * \overset{[N,10]}{(h^{(3)})} * \overset{[N,10]}{(1\text{-}h^{(3)})}$$

逐元素乘

$$\overset{[1,10]}{\frac{\partial E}{\partial b^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial b^{(3)}} = \overset{[1,N]}{\underset{\text{全1}}{I^{T}}}\ \overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

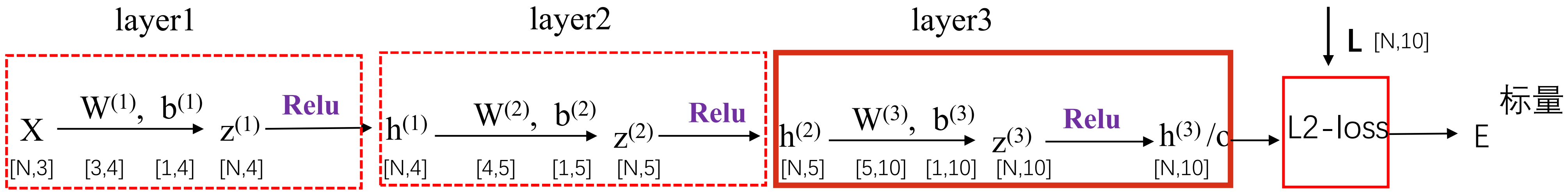
$$E = \frac{1}{2}(o-L)^2, \frac{\partial E}{\partial o} = o - L$$

关于z的函数

Relu 函数 $h = f(z) = \begin{cases} z & z > 0 \\ 0 & z \le 0 \end{cases} \quad h' = \frac{\partial f(z)}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases} = \text{de\_relu}(z)$

Sigmoid函数 $h = \frac{1}{1+e^{-z}}, h' = -\frac{-e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = h(1-h)$

领用梯度下降
进行参数$w^{(i)}$,$b^{(i)}$的更新



根据链式求导

$\frac{\partial E}{\partial z^{(3)}} \xleftarrow{\text{de-relu}}$ (o-L)

输入的转置　　误差

$$\overset{[5,10]}{\frac{\partial E}{\partial W^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial W^{(3)}} = \overset{[5,N]}{h^{(2)T}}\overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

$$\frac{\partial E}{\partial z^{(3)}} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z^{(3)}} = \overset{[N,10]}{(\text{o-L})} * \overset{[N,10]}{\text{de\_relu}(z^{(3)})}$$

逐元素乘

$$\overset{[1,10]}{\frac{\partial E}{\partial b^{(3)}}} = \frac{\partial E}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial b^{(3)}} = \overset{[1,N]}{I^{T}}\overset{[N,10]}{\frac{\partial E}{\partial z^{(3)}}}$$

全1

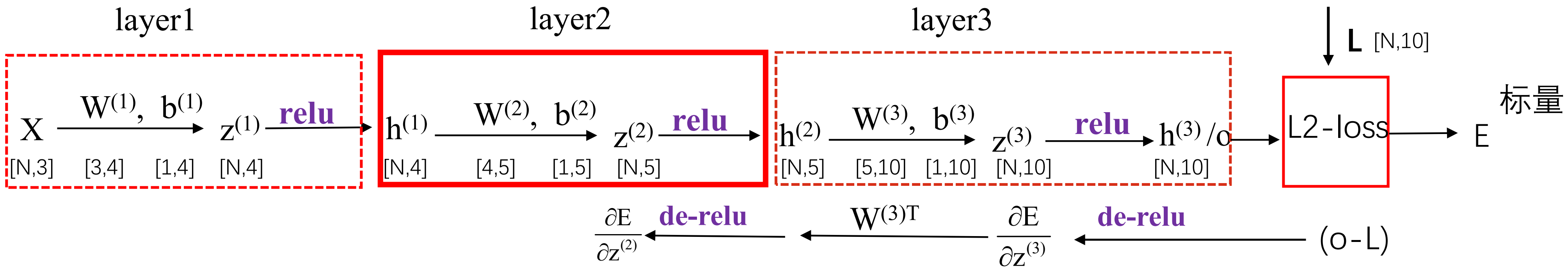
$$E = \frac{1}{2}(o-L)^2, \frac{\partial E}{\partial o} = o - L$$

关于z的函数

Relu 函数 $h = f(z) = \begin{cases} z & z > 0 \\ 0 & z \le 0 \end{cases}$ $h' = \frac{\partial f(z)}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z \le 0 \end{cases}$ =de_relu(z)

领用梯度下降
进行参数$w^{(i)}$,$b^{(i)}$的更新

Sigmoid函数 $h = \frac{1}{1+e^{-z}}, h' = -\frac{-e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = h(1-h)$



根据链式求导

[4,5] [4,N] [N,5]

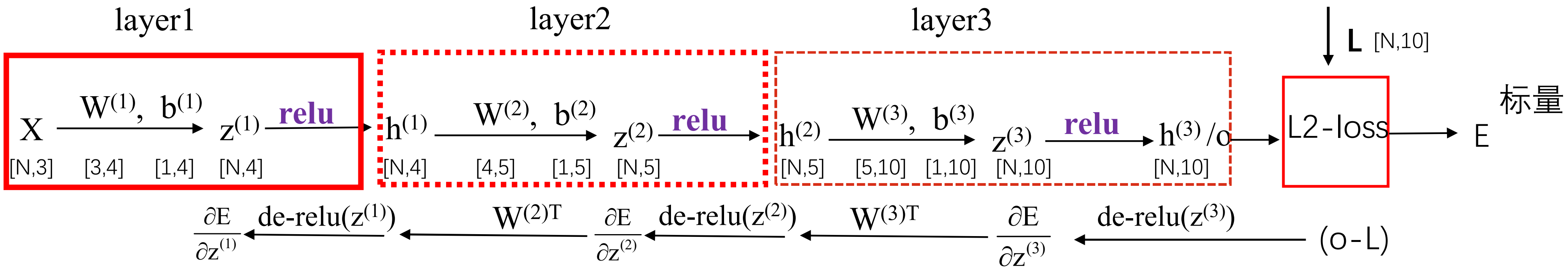
$$\frac{\partial E}{\partial W^{(2)}} = \frac{\partial E}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}} = h^{(1)T} \frac{\partial E}{\partial z^{(2)}}$$

[1,5] [1,N] [N,5]

$$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = I^{T} \frac{\partial E}{\partial z^{(2)}}$$

[N,5] [N,10] [10,5] [N,5]

$$\frac{\partial E}{\partial z^{(2)}} = \frac{\partial E}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial z^{(2)}} = \frac{\partial E}{\partial z^{(3)}} W^{(3)T} * \mathrm{de\_relu}(z^{2})$$



根据链式求导

$$\overset{[3,4]}{\frac{\partial E}{\partial W^{(1)}}} = \frac{\partial E}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial W^{(1)}} = \overset{[3,N]}{X^{T}}\ \overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}}$$

$$\overset{[1,4]}{\frac{\partial E}{\partial b^{(1)}}} = \frac{\partial E}{\partial z^{(1)}}\frac{\partial z^{(1)}}{\partial b^{(1)}} = \overset{[1,N]}{I^{T}}\ \overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}}$$

$$\overset{[N,4]}{\frac{\partial E}{\partial z^{(1)}}} = \frac{\partial E}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial h^{(1)}}\frac{\partial h^{(1)}}{\partial z^{(1)}} = \overset{[N,5]}{\frac{\partial E}{\partial z^{(2)}}}\ \overset{[5,4]}{W^{(2)T}} * \overset{[N,4]}{de\_relu(z^{(1)})}$$

代码修改

Sigmoid 前向 反向

```
import numpy as np

def sigmod(z):
    h = 1./(1+np.exp(-z))
    return h


def de_sigmoid(h):
    return h*(1-h)
```

无激活函数 前向，反向

```
def no_active(z):
    h = z
    return h

def de_no_active(h):
    return np.ones(h.shape)
```

Relu 激活函数

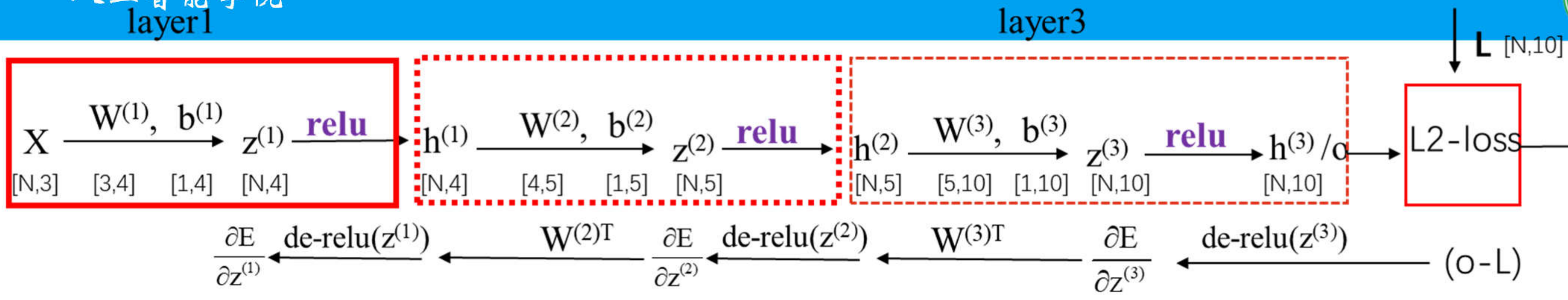
```
def relu(z):
    h = np.maximum(z, 0)
    return h

def de_relu(z,h):
    z[z <= 0] = 0
    z[z > 0] = 1.0
    return z
```

z表示激活函数的**输入**　**h** 表示激活函数的**输出**

修正

```
def sigmod(z):
    h = 1./(1+np.exp(-z))
    return h

def de_sigmoid(z,h):
    return h*(1-h)
```

```
def no_active(z):
    h = z
    return h

def de_no_active(z,h):
    return np.ones(h.shape)
```



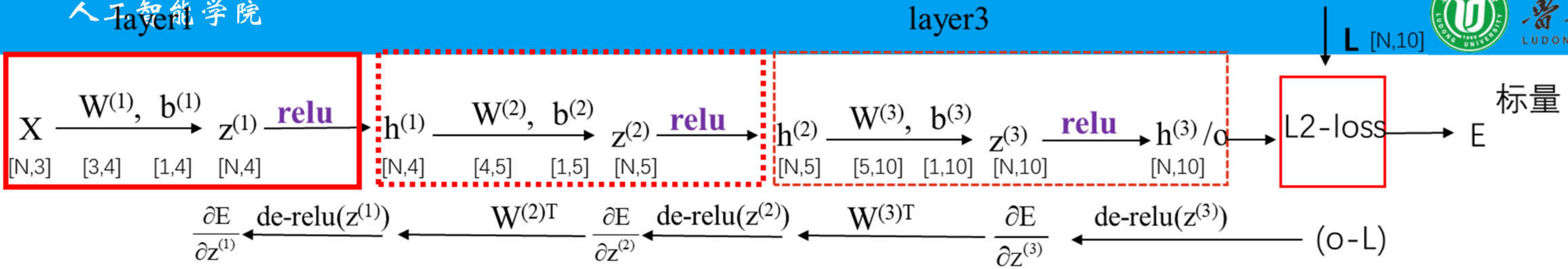
原版

```
# 返回每一层的输入
# 与最后一层的输出
def fead_forward(datas,layers):
    input_layers = []

    for i in range(len(layers)):
        layer = layers[i]
        if i ==0:
            inputs = datas
            z = np.dot(inputs,layer["w"]) + layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
        else:
            inputs = h
            z = np.dot(inputs,layer["w"])+ layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
    return input_layers,h
```

修改

```
# 返回每一层的输入
# 与最后一层的输出
def fead_forward(datas,layers):
    input_layers = []
    input_acfun = []
    for i in range(len(layers)):
        layer = layers[i]
        if i ==0:
            inputs = datas
            z = np.dot(inputs,layer["w"]) + layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
            input_acfun.append(z)
        else:
            inputs = h
            z = np.dot(inputs,layer["w"])+ layer["b"]
            h = layer['act_fun'](z)
            input_layers.append(inputs)
            input_acfun.append(z)
    return input_layers,input_acfun,h
```

对每一层网络增加保存**激活函数的输入z**




```python
# 进行参数更新更新
def updata_wb(datas,labs,layers, loss_fun,de_loss_fun,alpha=0.01):
    N,D = np.shape(datas)
    # 进行前馈操作
    inputs,input_acfun,output = fead_forward(datas,layers)
    # 计算 loss
    loss = loss_fun(output,labs)
    #从后向前计算
    deltas0 = de_loss_fun(output,labs)
    # 从后向前计算误差
    deltas =[]
    for i in range(len(layers)):
        index = -i-1
        if i ==0:
            h = output
            z = input_acfun[index]
            delta = deltas0*layers[index]["de_act_fun"](z,h)
        else:
            h = inputs[index+1]
            z = input_acfun[index]
            # print(layers[index]["de_act_fun"](z,h)[1])
            delta = np.dot(delta,layers[index+1]["w"].T)*layers[index]["de_act_fun"](z,h)

        deltas.insert(0,delta)
```

```python
# 利用误差 对每一层的权重进行修成
for i in range(len(layers)):
    # 计算 dw 与 db
    dw = np.dot(inputs[i].T,deltas[i])
    db = np.sum(deltas[i],axis=0,keepdims=True)
    # 梯度下降
    layers[i]["w"] = layers[i]["w"] - alpha*dw
    layers[i]["b"] = layers[i]["b"] - alpha*db

return layers,loss
```

测试函数

```
def test_accuracy(datas,labs_true,layers):
    _,_,output = fead_forward(datas,layers)
    lab_det = np.argmax(output,axis=1)
    labs_true = np.argmax(labs_true,axis=1)
    N_error = np.where(np.abs(labs_true-lab_det)>0)[0].shape[0]

    error_rate = N_error/np.shape(datas)[0]
    return error_rate
```

在MINIST数据集上的应用

加载数据

```python
import numpy as np
from NN_BP import *

def load_mnist(file_data,file_lab):
    # 加载训练数据
    data = np.load(file_data)
    lab = np.load(file_lab)
    N,D = np.shape(data)

    # 构造 one-hot 标签
    lab_onehot = np.zeros([N,10])
    for i in range(N):
        id = int(lab[i,0])
        lab_onehot[i,id]=1
    data = (data.astype(np.float)/255.0)
    return data,lab_onehot
```

定义网络

```python
if __name__=="__main__":

    # 加载训练数据
    train_data,train_lab_onehot=load_mnist("train_data.npy","train_lab.npy")
    N,D = np.shape(train_data)

    # 搭建网络
    # 定义网络结构
    list_num_hidden=[30,5,10]

    # list_act_funs =[sigmod,sigmod,sigmod]
    # list_de_act_funs=[de_sigmoid,de_sigmoid,de_sigmoid]
    # # 定义损失函数
    # loss_fun = loss_L2
    # de_loss_fun=de_loss_L2

    list_act_funs =[relu,relu,no_active]
    list_de_act_funs=[de_relu,de_relu,de_no_active]
    # 定义损失函数
    loss_fun = loss_CE
    de_loss_fun=de_loss_CE

    layers = bulid_net(D,list_num_hidden,
            list_act_funs,list_de_act_funs)
```

进行训练并测试

```
# 进行训练
n_epoch = 50
batchsize =20
N_batch = N//batchsize
for i in range(n_epoch):
    # 数据打乱
    rand_index  = np.random.permutation(N).tolist()
    # 每个batch 更新一下weight
    loss_sum =0
    for j in range(N_batch):
        index = rand_index[j*batchsize:(j+1)*batchsize]
        batch_datas = train_data[index]
        batch_labs = train_lab_onehot[index]
        layers,loss = updata_wb(batch_datas,batch_labs,layers,loss_fun,de_loss_fun,alpha=0.001)
        # print("epoch %d  batch %d  loss %.2f"%(i,j,loss/batchsize))
        loss_sum = loss_sum+loss

    error = test_accuracy(train_data,train_lab_onehot,layers)
    print("epoch %d  error  %.2f%%  loss_all %.2f"%(i,error*100,loss_sum/(N_batch*batchsize)))

np.save("model.npy",layers)

# 加载测试数据
test_data,test_lab_onehot=load_mnist("test_data.npy","test_lab.npy")
layers = np.load("model.npy",allow_pickle=True)

error = test_accuracy(test_data,test_lab_onehot,layers)
print("Accuarcy on Test Data %.2f %%"%((1-error)*100))
```

实验过程中会出现错误率很高且损失不下降的情况

```
  data = (data.astype(np.float)/255.0)
epoch 0  error  88.89%  loss_all 2.30
epoch 1  error  88.88%  loss_all 2.30
epoch 2  error  88.88%  loss_all 2.30
epoch 3  error  88.88%  loss_all 2.30
```

原因：与sigmoid不同，relu 激活函数对神经节点的输出不会进行约束，若干层迭代后输出的值过大

解决方法：网络初始化时采用较小的权重

```
def bulid_net(dim_in,list_num_hidden,
              list_act_funs,list_de_act_funs):
    layers=[]

    # 逐层的进行网络构建
    for i in range(len(list_num_hidden)):
        layer = {}

        # 定义每一层的权重
        if i ==0:
            # layer["w"]= 0.2*np.random.randn(dim_in,list_num_hidden[i])-0.1 # 用sigmoid激活函数
            layer["w"]= 0.01*np.random.randn(dim_in,list_num_hidden[i])  # 用relu 激活函数
        else:
            # layer["w"]= 0.2*np.random.randn(list_num_hidden[i-1],list_num_hidden[i])-0.1 # 用sigmoid激活函数
            layer["w"]= 0.01*np.random.randn(list_num_hidden[i-1],list_num_hidden[i]) # 用relu 激活函数

        # 定义每一层的偏置
        layer["b"] = 0.1*np.ones([1,list_num_hidden[i]])
        layer["act_fun"]= list_act_funs[i]
        layer["de_act_fun"]= list_de_act_funs[i]
        layers.append(layer)

    return layers
```

修改前

```
epoch 0   error  88.89%  loss_all 2.30
epoch 1   error  88.88%  loss_all 2.30
epoch 2   error  88.88%  loss_all 2.30
epoch 3   error  88.88%  loss_all 2.30
epoch 4   error  88.88%  loss_all 2.30
epoch 5   error  88.88%  loss_all 2.30
epoch 6   error  88.88%  loss_all 2.30
epoch 7   error  88.89%  loss_all 2.30
epoch 8   error  88.88%  loss_all 2.30
epoch 9   error  88.88%  loss_all 2.30
epoch 10  error  88.88%  loss_all 2.30
epoch 11  error  88.88%  loss_all 2.30
epoch 12  error  88.88%  loss_all 2.30
epoch 13  error  88.88%  loss_all 2.30
epoch 14  error  88.88%  loss_all 2.30
epoch 15  error  88.88%  loss_all 2.30
epoch 16  error  88.88%  loss_all 2.30
epoch 17  error  88.88%  loss_all 2.30
epoch 18  error  88.88%  loss_all 2.30
epoch 19  error  88.88%  loss_all 2.30
epoch 20  error  88.89%  loss_all 2.30
epoch 21  error  88.88%  loss_all 2.30
```

```
epoch 45  error  88.88%  loss_all 2.30
epoch 46  error  88.88%  loss_all 2.30
epoch 47  error  88.88%  loss_all 2.30
epoch 48  error  88.88%  loss_all 2.30
epoch 49  error  88.88%  loss_all 2.30
Accuarcy on Test Data 12.07 %
```

修改后

```
epoch 0   error  88.88%  loss_all 2.30
epoch 1   error  26.41%  loss_all 1.32
epoch 2   error  11.39%  loss_all 0.54
epoch 3   error  8.01%  loss_all 0.34
epoch 4   error  6.30%  loss_all 0.26
epoch 5   error  5.56%  loss_all 0.22
epoch 6   error  5.27%  loss_all 0.19
epoch 7   error  5.26%  loss_all 0.17
epoch 8   error  4.84%  loss_all 0.16
epoch 9   error  3.96%  loss_all 0.15
epoch 10  error  3.92%  loss_all 0.14
epoch 11  error  3.16%  loss_all 0.13
epoch 12  error  3.08%  loss_all 0.12
epoch 13  error  3.19%  loss_all 0.12
epoch 14  error  3.03%  loss_all 0.11
epoch 15  error  2.75%  loss_all 0.11
epoch 16  error  2.72%  loss_all 0.10
epoch 17  error  2.60%  loss_all 0.10
epoch 18  error  2.20%  loss_all 0.09
epoch 19  error  2.46%  loss_all 0.09
epoch 20  error  2.50%  loss_all 0.09
epoch 21  error  2.35%  loss_all 0.08
epoch 22  error  2.48%  loss_all 0.08
epoch 23  error  2.35%  loss_all 0.08
epoch 24  error  2.06%  loss_all 0.08
epoch 25  error  2.04%  loss_all 0.08
epoch 26  error  1.72%  loss_all 0.07
epoch 27  error  2.10%  loss_all 0.07
epoch 28  error  1.65%  loss_all 0.07
epoch 29  error  1.89%  loss_all 0.07
epoch 30  error  1.89%  loss_all 0.07
epoch 31  error  1.63%  loss_all 0.06
```

```
epoch 40  error  1.45%  loss_all 0.05
epoch 41  error  1.67%  loss_all 0.05
epoch 42  error  1.49%  loss_all 0.05
epoch 43  error  1.44%  loss_all 0.05
epoch 44  error  1.13%  loss_all 0.05
epoch 45  error  1.29%  loss_all 0.05
epoch 46  error  1.08%  loss_all 0.04
epoch 47  error  1.27%  loss_all 0.05
epoch 48  error  1.05%  loss_all 0.04
epoch 49  error  0.97%  loss_all 0.04
Accuarcy on Test Data 96.18 %
```