> 由于时间关系，文档写的不是很完整。
>
> ——李新浩 2021.11.15

反向传播算法实现

如图所示，为简化描述，我们用一个三元组$(in_i, out_i, bias_i)$来描述第i层layer，分表表示`in_feature`、`out_feature`、`bias`，使用X_i表示每层的输入`inputs[i]`，Y_i表示每层的输出`outputs[i]`。

前向传播

原理

正向传播过程十分简单，假设n为样本数，输入为$n \times 2$的输入矩阵X_0，那么正向传播过程为：

1. 首先经过第0层的权重矩阵：

$$X_0 W_0 = X_i \begin{pmatrix} w1 & w3 \\ w2 & w4 \end{pmatrix} + \begin{pmatrix} b1 \\ b2 \end{pmatrix} = [X_0, \text{ones(n, 1)}] \begin{pmatrix} w1 & w3 \\ w2 & w4 \\ b1 & b2 \end{pmatrix} \tag{1}$$

2. 再经过激活函数得到第0层的输出也就是第1层的输入：

$$X_1 = Y_0 = \sigma_0(X_0 W_0) \tag{2}$$

3. 第1层再经过同样的步骤得到最终网络的输出：

$$\begin{gathered} X_1 W_1 = X_1 \begin{pmatrix} w5 & w7 \\ w6 & w8 \end{pmatrix} + \begin{pmatrix} b3 \\ b4 \end{pmatrix} = [X_1, \text{ones(n, 1)}] \begin{pmatrix} w5 & w7 \\ w6 & w8 \\ b3 & b4 \end{pmatrix} \\ Y_1 = \sigma_1(X_1 W_1) \end{gathered} \tag{3}$$

核心代码

```python
for i, layer in enumerate(self.layers):
    if layer.bias:  # 添加偏置项
        x = np.concatenate(
            [x, np.ones((len(x), 1))], axis=1)
        x = np.matmul(x, self.w[i])
        x = layer.activation(x)
```

实现细节

设样本数为n，则开始有$shape$为(n, in_0)的网络输入X_{input}，则有：

$$Y_0 = \text{activation}(W_0 X_0) \tag{4}$$

$$\begin{cases} X_0.shape = (n, in_0 + 1), W_0.shape = (in_0 + 1, out_0) & bias_0 = true \\ X_0.shape = (n, in_0), W_0.shape = (in_0, out_0) & bias_0 = false \end{cases} \tag{5}$$

$$Y_0.shape = (n, out_0) \tag{6}$$

显然后面的层也是一样的：

$$X_i = Y_{i-1}$$

$$Y_i = \text{activation}(W_i X_i) \tag{7}$$

$$\begin{cases} X_i.\,shape = (n, in_i + 1), W_i.\,shape = (in_i + 1, out_i) & bias_i = true \\ X_i.\,shape = (n, in_i), W_i.\,shape = (in_i, out_i) & bias_i = false \end{cases} \tag{8}$$

$$Y_i.\,shape = (n, out_i) \tag{9}$$

反向传播

原理

我们用δ_i表示第i层的反向传播中间变量delta向量（向量里面也从0开始数，δ_1^0表示δ_1第一个元素），用$S(\cdot)$表示激活函数，x表示网络输入向量，网络输入输出表示为In_i、Out_i，即$Out_i = S(In_i)$:

1. 首先计算输出层（我们这里就是第1层）delta向量δ_1:

$$\begin{pmatrix} \delta_1^0 \\ \delta_1^1 \end{pmatrix} = \begin{pmatrix} \partial loss / \partial Out_1^0 \\ \partial loss / \partial Out_1^1 \end{pmatrix} \odot \begin{pmatrix} \partial S(In_1^0)/In_1^0 \\ \partial S(In_1^1)/In_1^1 \end{pmatrix} \tag{10}$$

2. 然后计算隐藏层（我们这里就是第0层）delta向量δ_0:

$$\begin{pmatrix} \delta_0^0 \\ \delta_0^1 \end{pmatrix} = \left(W_1^{\text{without bias}} \begin{pmatrix} \delta_1^0 \\ \delta_1^1 \end{pmatrix} \right) \odot \begin{pmatrix} \partial S(In_0^0)/In_0^0 \\ \partial S(In_0^1)/In_0^1 \end{pmatrix} \tag{11}$$

3. 有了delta向量后就能轻而易举地算出梯度进而算出权重变化值:

$$\Delta W_1 = \text{learning rate} \cdot \begin{pmatrix} Out_0^0 \\ Out_0^1 \\ 1 \end{pmatrix} (\delta_1^0, \delta_1^1)^T \tag{12}$$

$$\Delta W_0 = \text{learning rate} \cdot \begin{pmatrix} x^0 \\ x^1 \\ 1 \end{pmatrix} (\delta_0^0, \delta_0^1)^T \tag{13}$$

4. 显然，对于超过2层的网络，中间层数对应的delta计算和权重更新如下:

$$\begin{pmatrix} \delta_i^0 \\ \delta_i^1 \end{pmatrix} = \left(W_{i+1}^{\text{without bias}} \begin{pmatrix} \delta_{i+1}^0 \\ \delta_{i+1}^1 \end{pmatrix} \right) \odot \begin{pmatrix} \partial S(In_i^0)/In_i^0 \\ \partial S(In_i^1)/In_i^1 \end{pmatrix} \tag{14}$$

$$\Delta W_i = \text{learning rate} \cdot \begin{pmatrix} Out_{i-1}^0 \\ Out_{i-1}^1 \\ 1 \end{pmatrix} (\delta_i^0, \delta_i^1)^T \tag{15}$$

核心代码

```
delta = grad_loss * \
    self.layers[-1].grad_activation(inputs[-1])  # grad_loss * grad_S
tmp_output = outputs[-2]
if self.layers[-1].bias:  # 偏置权重也要更新
    tmp_output = np.concatenate([tmp_output, np.ones((1, 1))], axis=0)
# 马上算grad是为了不存delta数组
self.grad_W[-1] += np.matmul(tmp_output, delta.T)
for i in range(len(self.layers)-2, -1, -1):
    if self.layers[i+1].bias:  # 更新delta时不算偏置那项
        delta = np.matmul(self.W[i+1][:-1, :], delta)
```

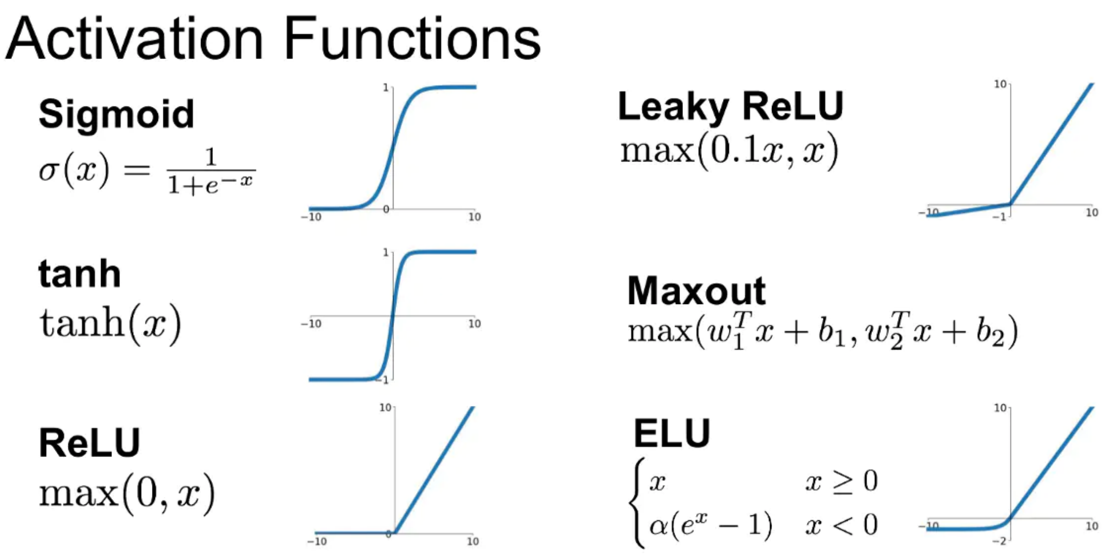
```
        else:
            delta = np.matmul(self.W[i+1], delta)
        delta *= self.layers[i].grad_activation(inputs[i])
        # 计算梯度
        tmp_output = outputs[i]  # 实际是outputs_{i-1}
        if self.layers[i].bias:  # 偏置权重也要更新
            tmp_output = np.concatenate(
                [tmp_output, np.ones((1, 1))], axis=0)
        # 马上算grad是为了不存delta数组
        self.grad_W[i] += np.matmul(tmp_output, delta.T)
```

辅助函数

独热码

```
def one_hot(X):
    assert len(X.shape) == 1, "输入必须是数值型向量"
    X = X - X.min()  # 从0开始数
    return np.eye(X.max()+1)[X]
```

激活函数



sigmoid

比较经典的一个激活函数，也是逻辑回归的一个联系函数：

$$\mathrm{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{16}$$

其导数/梯度计算较为简单：

$$\frac{\mathrm{d}\,\mathrm{sigmoid}(x)}{\mathrm{d}x} = \frac{e^{-x}}{(1+e^{-x})^2} = \mathrm{sigmoid}(x)(1-\mathrm{sigmoid}(x)) \tag{17}$$

代码实现也很简单：

```python
def sigmoid(X):
    return 1 / (1 + np.exp(-X))

def grad_sigmoid(X):
    return sigmoid(X) * (1.0 - sigmoid(X))
```

softmax

一般用在输出层，使logit变成probablity：

$$\mathrm{softmax}(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}} \tag{18}$$

```python
def softmax(X):
    exps = np.exp(X)
    return exps / np.sum(exps)
```

但是指数计算容易溢出，一种简单的解决方式是直接在分子分母上都乘一个较小的数值C：

$$\begin{aligned}\mathrm{softmax}(x_j) &= \frac{Ce^{x_j}}{\sum_{i=1}^{N} Ce^{x_i}} \\ &= \frac{e^{x_j+\log C}}{\sum_{i=1}^{N} e^{x_i+\log C}} \\ &= \frac{e^{x_j+D}}{\sum_{i=1}^{N} e^{x_i+D}}\end{aligned} \tag{19}$$

往往可以取$D = -\max(x_1, \cdots, x_N)$：

```python
def stable_softmax(X):
    shift_X = X - np.max(X)
    exps = np.exp(shift_X)
    return exps / np.sum(exps)
```

另一种方式是在外面加一个log，然后再求exp：

$$\begin{aligned}\log\left(\mathrm{softmax}(x_j)\right) &= \log\left(\frac{e^{x_j}}{\sum_{i=1}^{N} e^{x_i}}\right) \\ &= \log e^{x_j} - \log\left(\sum_{i=1}^{N} e^{x_i}\right) \\ &= x_j - \log\left(\sum_{i=1}^{N} e^{x_i}\right)\end{aligned} \tag{20}$$

可以看到，logSoftmax数值上相对稳定一些。`Softmax_Cross_Entropy` 里面也是这么实现的。

最后是计算softmax的梯度，整个softmax里面的操作都是可微的，所以梯度计算也非常简单：

$$\begin{aligned}\frac{\partial\, \mathrm{softmax}(x_j)}{\partial x_j} &= \frac{\partial e^{x_j} / \sum_{i=1}^{N} e^{x_i}}{\partial x_j} \\ = \frac{e^{x_j}\sum_{i=1}^{N} e^{x_i} - e^{2x_j}}{(\sum_{i=1}^{N} e^{x_i})^2} &= \frac{e^{x_j}(\sum_{i=1}^{N} e^{x_i} - e^{x_j})}{(\sum_{i=1}^{N} e^{x_i})^2} \\ &= \mathrm{softmax}(x_j)(1 - \mathrm{softmax}(x_j))\end{aligned} \tag{21}$$

$$= \text{softmax}(x_j)(1 - \text{softmax}(x_j))$$

可以发现和sigmoid函数具有相同的性质，值得一提的是由于$\text{softmax}(x_j)$的值不止受x_j影响，还受其他神经元的影响，因此对于$i \neq j$，有：

$$\begin{aligned}\frac{\partial \,\text{softmax}(x_j)}{\partial x_i} &= \frac{-e^{x_i}e^{x_j}}{(\sum_{i=1}^{N} e^{x_i})^2} \\ &= -\text{softmax}(x_i)\text{softmax}(x_j)\end{aligned} \tag{22}$$

不过这一部分梯度在反向传播算法中是用不上的，因此我们计算梯度向量可以实现为：

```python
def grad_softmax(X):
    return softmax(X) * (1.0 - softmax(X))
```

ReLU

最常用的激活函数之一，形式相当简单：

$$\text{ReLu}(x) = \max(0, x) \tag{23}$$

```python
def relu(X):
    return np.maximum(X, np.zeros_like(X))

def grad_relu(X):
    return (X > 0).astype(np.float32)
```

损失函数

MSE

最简单的损失函数，直接看代码吧：

```python
def mse(y_pred, y_true, method='mean'):
    r'''
    均方差
    y_pred.shape = (n_samples, n_output)
    y_true.shape = (n_samples, n_output) 分类的话需要是独热码
    '''
    assert y_pred.shape == y_true.shape, "y_pred和y_true的shape不相等!"
    if method is None:
        return (y_pred - y_true)**2
    elif method == 'mean':
        return np.mean((y_pred - y_true)**2, axis=0)
    elif method == 'sum':
        return np.sum((y_pred - y_true)**2, axis=0)
    else:
        assert False, f"不支持method={method}!"


def grad_mse(y_pred, y_true):
    r'''
    返回均方根误差梯度向量
    '''
    assert y_pred.shape == y_true.shape, "y_pred和y_true的shape不相等!"
    return -(y_true - y_pred)
```

交叉熵

> y表示真值，p表示预测值

二分类交叉熵

二分类交叉熵通常针对于输出层只有一个神经元的网络：

$$L=-(y\log p+(1-y)\log{(1-p)}) \tag{24}$$

求导也很简单：

$$\frac{\partial L}{\partial p}=-(\frac{y}{p}-\frac{1-y}{1-p}) \tag{25}$$

```
def binary_cross_entropy(y_pred, y_true):
    return -(y_true*np.log(y_pred)+(1-y_true)*np.log(1-y_pred))

def grad_binary_cross_entropy(y_pred, y_true):
    return -(y_true/y_pred - (1-y_true)/(1-y_pred))
```

多分类交叉熵

多分类交叉熵则是最常见的交叉熵定义：

$$L=-\frac{1}{N}\sum_{i=1}^{N}(y_i\log p_i+(1-y_i)\log{(1-p_i)}) \tag{26}$$