

BACKPROPAGATION

一、内容

本部分将实现Dense Layer、Activation Function和Loss的反向传播。

二、代码

一、Dense Layer

公式

$$y = wx + b$$

$$\frac{\partial y}{\partial w} = x$$

$$\frac{\partial y}{\partial x} = w$$

$$\frac{\partial y}{\partial b} = 1$$

其中 x 是输入向量， w 是权重， b 是偏置， y 是Dense Layer层是输出向量， b 和 w 已经在初始化时保存，所以在前向传播中要将 x 保存在Dense Layer的属性中，**注意：1和 w 一样是一个矩阵，但大小不一样。**相关代码如下：

实现

```
def forward(self, input):
    # 因为要增加backward方法,
    # Layer_Dense的输出对输入 (input) 的偏导是self.weight,
    # 而Layer_Dense的输出对self.weight的偏导是输入 (input)
    # 所以要在forward中增加self.input属性
    self.input = input #self.input是相对前面代码版本中新加入的
    self.output = np.dot(input, self.weight) + self.bias
```

公式

$$loss = f(y)$$

$$\frac{\partial loss}{\partial y} = dvalue$$

$$\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial y} \frac{\partial y}{\partial w} = dvalue * \frac{\partial y}{\partial w} = dvalue * x$$

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial y} \frac{\partial y}{\partial x} = dvalue * \frac{\partial y}{\partial x} = dvalue * w$$

$$\frac{\partial loss}{\partial b} = \frac{\partial loss}{\partial y} \frac{\partial y}{\partial b} = dvalue * \frac{\partial y}{\partial b} = dvalue * 1$$

其中的dvalue通过下一层的反向传播求得，并作为这一层backward方法的参数，所以dvalue在该层中是已知的，只需通过代码实现求 $\frac{\partial y}{\partial w}$ 和 $\frac{\partial y}{\partial x}$ ，即 x 和 w ，代码如下：

实现

```
def backward(self, dvalue):
    # dvalue是loss对下一层 (Activation) 的输入 (input) 的导数,
    # 也就是loss对这一层 (Layer_Dense) 的输出 (output) 的导数,
    # 这里会用到链式法则

    # 在本层中，希望求得的是loss对这一层 (Layer_Dense) 的self.weight的导数
    # 这便找到了self.weight优化的方向 (negative gradient direction)

    # 这里要考虑到self.dweight的大小要与self.weight一致，因为方便w - lr * dw公式进行优化
    # 假设input只有一个sample，大小为1xa，weight大小为axb，则output大小为1xb，
    # 因为loss是标量，所以dvalue = dloss/doutput大小即为output的大小(1xb)，
    # 所以dweight的大小为(1xa).T * (1xb) = axb，大小和weight一致。
    # 注意：当input有多个sample时（一个矩阵输入），则dweight为多个axb矩阵相加。
```

```

self.dweight = np.dot(self.input.T, dvalue)

# 在本层中，希望求得的是loss对这一层（Layer_Dense）的self.input的导数
# 以便作为下一层的backward方法中的dvalue参数，

# 因为loss是标量，所以dinput大小即为input的大小(1xa)，
# dvalue = dloss/doutput大小即为output的大小(1xb)，
# weight大小为axb
# 所以1xa = (1xb) * (axb).T
self.dinput = np.dot(dvalue, self.weight.T)

# 像self.dinput一样，self.dbias可以通过矩阵乘法实现，
# self.dbias = np.dot( dvalue, np.ones( ( len(self.bias), len(self.bias) ) ) )
# 但有更快更简单的实现
self.dbias = np.sum(dvalue, axis=0, keepdims=True)# 此处不要keepdims=True也行，因为按0维相加还是行向量

```

二、ReLU

公式

$$y = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

$$\frac{dy}{dx} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

$$loss = f(y)$$

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial y} \frac{\partial y}{\partial x} = dvalue * \frac{\partial y}{\partial x} = \begin{cases} dvalue, & x > 0 \\ 0, & x < 0 \end{cases}$$

从矩阵的角度看 $\frac{\partial y}{\partial x}$ 是一个对角方阵，对角线上的值为dvalue或0，但实际并不用矩阵乘法实现

实现

```
def backward(self, dvalue):
    # self.input和self.output形状是一样的
    # 那么dinput大小=doutput大小=dvalue大小
    # 可以用mask来更快实现，而不用矩阵运算
    self.dinput = dvalue.copy()
    self.dinput[self.input < 0] = 0
```

三、Categorical Cross-Entropy loss

公式

$$L_i = - \sum_j y_{i,j} \log(\hat{y}_{i,j})$$

其中 L_i 表示样本损失值， i 表示集合中的第 i 个样本， j 表示标签索引， y 表示目标值， \hat{y} 表示预测值。

$$\begin{aligned} \frac{\partial L_i}{\partial \hat{y}_{i,j}} &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[- \sum_j y_{i,j} \log(\hat{y}_{i,j}) \right] = - \sum_j y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \log(\hat{y}_{i,j}) = \\ &= - \sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} = - \sum_j y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 = - \sum_j \frac{y_{i,j}}{\hat{y}_{i,j}} = - \frac{y_{i,j}}{\hat{y}_{i,j}} \end{aligned}$$

实现

```
def backward(self, y_pred, y_true):
    n_sample = len(y_true)
    if len(y_true.shape) == 2: # 标签是onehot的编码
        label = y_true
    elif len(y_true.shape) == 1: # 只有一个类别标签
        # 将标签改成onehot的编码
        label = np.zeros((n_sample, len(y_pred[0])))
        label[range(n_sample), y_true] = 1
    self.dinput = - label / y_pred
    # 每个样本除以n_sample，因为在优化的过程中要对样本求和
    self.dinput = self.dinput / n_sample
```

四、Softmax

公式

Softmax函数是一种将j个实数向量转换为j个可能结果的概率分布的函数。索引表示当前样本，索引表示当前样本中的当前输出， $S_{i,j}$ 表示j个可能结果的概率。

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}}$$
$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}}}{\frac{\partial}{\partial z_{i,k}} \sum_{l=1}^L e^{z_{i,l}}}$$

当 $j = k$ ，推导如下：

$$\begin{aligned} \frac{\partial S_{i,j}}{\partial z_{i,k}} &= \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^L e^{z_{i,l}}}{[\sum_{l=1}^L e^{z_{i,l}}]^2} = \\ &= \frac{e^{z_{i,j}} \cdot \sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^L e^{z_{i,l}}]^2} = \frac{e^{z_{i,j}} \cdot (\sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,k}})}{\sum_{l=1}^L e^{z_{i,l}} \cdot \sum_{l=1}^L e^{z_{i,l}}} = \\ &= \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}} \cdot \frac{\sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,k}}}{\sum_{l=1}^L e^{z_{i,l}}} = \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}} \cdot \left(\frac{\sum_{l=1}^L e^{z_{i,l}}}{\sum_{l=1}^L e^{z_{i,l}}} - \frac{e^{z_{i,k}}}{\sum_{l=1}^L e^{z_{i,l}}} \right) = \\ &= S_{i,j} \cdot (1 - S_{i,k}) \end{aligned}$$

当 $j \neq k$ ，推导如下：

$$\begin{aligned}
\frac{\partial S_{i,j}}{\partial z_{i,k}} &= \frac{\partial \frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}}}{\partial z_{i,k}} = \frac{\frac{\partial}{\partial z_{i,k}} e^{z_{i,j}} \cdot \sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,j}} \cdot \frac{\partial}{\partial z_{i,k}} \sum_{l=1}^L e^{z_{i,l}}}{[\sum_{l=1}^L e^{z_{i,l}}]^2} = \\
&= \frac{0 \cdot \sum_{l=1}^L e^{z_{i,l}} - e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^L e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{[\sum_{l=1}^L e^{z_{i,l}}]^2} = \frac{-e^{z_{i,j}} \cdot e^{z_{i,k}}}{\sum_{l=1}^L e^{z_{i,l}} \cdot \sum_{l=1}^L e^{z_{i,l}}} = \\
&= -\frac{e^{z_{i,j}}}{\sum_{l=1}^L e^{z_{i,l}}} \cdot \frac{e^{z_{i,k}}}{\sum_{l=1}^L e^{z_{i,l}}} = -S_{i,j} \cdot S_{i,k}
\end{aligned}$$

综上有：

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ -S_{i,j} \cdot S_{i,k} & j \neq k \end{cases}$$

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = \begin{cases} S_{i,j} \cdot (1 - S_{i,k}) & j = k \\ S_{i,j} \cdot (0 - S_{i,k}) & j \neq k \end{cases}$$

$$\delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial S_{i,j}}{\partial z_{i,k}} = S_{i,j} \cdot (\delta_{j,k} - S_{i,k}) = S_{i,j} \delta_{j,k} - S_{i,j} S_{i,k}$$

$$\frac{\partial loss}{\partial z_{i,k}} = \frac{\partial loss}{\partial S_{i,j}} \frac{\partial S_{i,j}}{\partial z_{i,k}}$$

实现

```
def backward(self, dvalue):
    # input和output大小相同都为1xa,
    # loss是标量, 那么dinput和doutput (即dvalue) 大小相同都为1xa,
    # output对input的导数为一个axa的方阵

    # 相同大小的空矩阵
    self.dinput = np.empty_like(dvalue)
    # 对每个sample (每一行) 循环
    for each, (single_output, single_dvalue) in enumerate(zip(self.output, dvalue)):
        # 这里是(1xa) * (axa) = 1xa是行向量
        # 这里要先将1xa向量变为1xa矩阵
        # 因为向量没有转置 (.T操作后还是与原来相同),
        # np.dot接收到向量后, 会调整向量的方向, 但得到的还是向量 (行向量), 就算得到列
        # 向量也会表示成行向量
        # np.dot接收到1xa矩阵后, 要考虑前后矩阵大小的匹配, 不然要报错, 最后得到的还是矩
        # 阵

        single_output = single_output.reshape(1, -1)
        jacobian_matrix = np.diagflat(single_output) -
        np.dot(single_output.T, single_output)
        # 因为single_dvalue是行向量, dot运算会调整向量的方向
        # 所以np.dot(single_dvalue, jacobian_matrix)和np.dot(jacobian_matrix,
        single_dvalue)

        # 得到的都是一个行向量, 但两都的计算方法不同, 得到的值也不同
        # np.dot(jacobian_matrix, single_dvalue)也是对的, 这样得到的才是行向量,
        # 而不是经过dot将列向量转置成行向量
        self.dinput[each] = np.dot(jacobian_matrix, single_dvalue)
```

五、Sigmoid

公式

$$\sigma_{i,j} = \frac{1}{1 + e^{-z_{i,j}}}$$

其中 $z_{i,j}$ 表示这个激活函数的输入, $\sigma_{i,j}$ 表示单个输出值。索引 i 表示当前样本, 索引 j 表示当前样本中的当前输出。 $\sigma_{i,j}$ 可理解成对第 j 对类别, 例如猫狗分类中狗类别的confidence(置信度)。当然, 一个模型可能要对多对类别分类, 例如: 高矮、胖瘦等。Sigmoid用于二分类

$$\begin{aligned}
\sigma_{i,j} &= \frac{1}{1 + e^{-z_{i,j}}} \rightarrow \sigma'_{i,j} = \frac{d}{dz_{i,j}} \left[\frac{1}{1 + e^{-z_{i,j}}} \right] = \frac{d}{dz_{i,j}} (1 + e^{-z_{i,j}})^{-1} = \\
&= -1 \cdot (1 + e^{-z_{i,j}})^{-1-1} \cdot \frac{d}{dz_{i,j}} (1 + e^{-z_{i,j}}) = -(1 + e^{-z_{i,j}})^{-2} \cdot \left(\frac{d}{dz_{i,j}} 1 + \frac{d}{dz_{i,j}} e^{-z_{i,j}} \right) = \\
&= -(1 + e^{-z_{i,j}})^{-2} \cdot (0 + e^{-z_{i,j}} \cdot \frac{d}{dz_{i,j}} [-z_{i,j}]) = \\
&= -(1 + e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1 \cdot \frac{d}{dz_{i,j}} z_{i,j})) = -(1 + e^{-z_{i,j}})^{-2} \cdot (e^{-z_{i,j}} \cdot (-1)) = \\
&= -(1 + e^{-z_{i,j}})^{-2} \cdot (-e^{-z_{i,j}}) = (1 + e^{-z_{i,j}})^{-2} \cdot e^{-z_{i,j}} = \\
&= \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})^2} = \frac{e^{-z_{i,j}}}{(1 + e^{-z_{i,j}})(1 + e^{-z_{i,j}})} = \frac{1}{1 + e^{-z_{i,j}}} \cdot \frac{e^{-z_{i,j}}}{1 + e^{-z_{i,j}}} = \\
&= \frac{1}{1 + e^{-z_{i,j}}} \cdot \frac{1 + e^{-z_{i,j}} - 1}{1 + e^{-z_{i,j}}} = \frac{1}{1 + e^{-z_{i,j}}} \cdot \left(\frac{1 + e^{-z_{i,j}}}{1 + e^{-z_{i,j}}} - \frac{1}{1 + e^{-z_{i,j}}} \right) = \\
&= \frac{1}{1 + e^{-z_{i,j}}} \cdot \left(1 - \frac{1}{1 + e^{-z_{i,j}}} \right) = \sigma_{i,j} \cdot (1 - \sigma_{i,j})
\end{aligned}$$

$$\frac{\partial loss}{\partial z_{i,k}} = \begin{cases} \frac{\partial loss}{\partial \sigma_{i,j}} \frac{\partial \sigma_{i,j}}{\partial z_{i,k}}, j = k \\ 0, j \neq k \end{cases}$$

k 取一个固定值，那么 j 每取一个值， $\frac{\partial loss}{\partial z_{i,k}}$ 都是标量；而 $\frac{\partial loss}{\partial z_{i,*}}$ 就是个行向量， $\frac{\partial \sigma_{i,*}}{\partial z_{i,*}}$ 是一个对角方阵。

这里可以用矩阵计算，但有更简单的方法，实现如下：

实现

```
def backward(self, dvalue):
    # 这里也可以用矩阵计算，但dinput、dvalue、output大小相同，
    # 可以直接按元素对应相乘。
    self.dinput = dvalue * self.output * (1 - self.output)
```


六、Binary Cross-Entropy loss

公式

$$\begin{aligned} L_{i,j} &= (y_{i,j})(-\log(\hat{y}_{i,j})) + (1 - y_{i,j})(-\log(1 - \hat{y}_{i,j})) = \\ &= -y_{i,j} \cdot \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j}) \end{aligned}$$

其中, $\hat{y}_{i,j}$ 是第 j 对二进制输出。

$$\begin{aligned} \frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} &= \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j})] = \\ &= \frac{\partial}{\partial \hat{y}_{i,j}} [-y_{i,j} \cdot \log(\hat{y}_{i,j})] + \frac{\partial}{\partial \hat{y}_{i,j}} [-(1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j})] = \\ &= -y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \log(1 - \hat{y}_{i,j}) = \\ &= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [1 - \hat{y}_{i,j}] = \\ &= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \left(\frac{\partial}{\partial \hat{y}_{i,j}} 1 - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} \right) = \\ &= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot 1 - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot (0 - 1) = \\ &= -\frac{y_{i,j}}{\hat{y}_{i,j}} + \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} = -\left(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \right) \end{aligned}$$

由于一个模型可以包含多个二进制输出, 因此在单个输出上计算的损失将组成一个损失向量, 其中每个输出都有一个值。需要的是一个样本损失, 需要计算所有这些来自单个样本的损失的平均。

$$L_i = \frac{1}{J} \sum_j L_{i,j}$$

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}} \left[\frac{1}{J} \sum_j L_{i,j} \right] = \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} = \frac{1}{J} \cdot 1 = \frac{1}{J}$$

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = \frac{1}{J} \cdot \left(-\left(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \right) \right) = -\frac{1}{J} \cdot \left(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \right)$$

实现

```
def backward(self, y_pred, y_true):
    # 样本个数
    n_sample = len(y_true)
    # 这里要特别注意，书上都没有写明
    # 当只有一对二进制类别时，y_pred大小为(n_sample, 1), y_ture大小为(n_sample,)
    # (n_sample,)和(n_sample, 1)一样都可以广播，只是(n_sample,)不能转置
    # 所以下面的loss大小会变成(n_sample, n_sample)
    # 当有二对二进制类别时，y_pred大小为(n_sample, 2), y_ture大小为(n_sample, 2)
    if len(y_true.shape) == 1: # y_true是个行向量
        y_true = y_true.reshape(-1, 1)
    # 注意：BinaryCrossentropy之前都是Sigmoid函数
    # Sigmoid函数很容易出现0和1的输出
    # 所以以1e-7为左边界
    # 另一个问题是将置信度向1移动，即使是非常小的值，
    # 为了防止偏移，右边界为1 - 1e-7
    y_pred = np.clip(y_pred, 1e-7, 1 - 1e-7)
    self.dinput = - y_true / y_pred + (1 - y_true) / (1 - y_pred)
    # 每个样本除以n_sample，因为在优化的过程中要对样本求和
    self.dinput = self.dinput / n_sample
```