《深度学习》实验 4 讲解

多层感知机/全连接层

岳锦鹏
10213903403

2024 年 4 月 11 日

目录

首先逐个观察每个填空的部分需要完成哪些内容。

可以看到需要完成 ReLU 的反向传播过程。

```
class Relu:
    def __init__(self):
        self.mem = {}

    def forward(self, x):
        self.mem['x'] = x
        return np.where(x > 0, x, np.zeros_like(x))

    def backward(self, grad_y):
        '''
        grad_y: same shape as x
        '''

        # ==========
        # todo '''请完成激活函数的梯度后传'''
        # ==========
```

对于主要的模型部分，需要完成计算损失。

```
def compute_loss(self, log_prob, labels):
    '''
    log_prob is the predicted probabilities
    labels is the ground truth
    Please return the loss
    '''

    # ==========
    # todo '''请完成多分类问题的损失计算 损失为: 交叉熵损失 + L2 正则项'''
    # ==========
```

按照给定的网络结构完成前向传播过程。

```
def forward(self, x):
    '''
    x is the input features
    Please return the predicted probabilities of x
    '''

    # ==========
    # todo '''请搭建一个 MLP 前馈神经网络 补全它的前向传播 MLP 结构为 FFN --> RELU --> FFN --> Softmax'''
    # ==========
```

完成主模型的后向传播，注意这里可以使用其中各层的反向传播函数。

```
def backward(self, label):
    '''
    label is the ground truth
    Please compute the gradients of self.W1 and self.W2
    '''

    # ==========
    # todo '''补全该前馈神经网络的后向传播算法'''
    # ==========
```

更新参数，这里要注意不要忘记正则项的损失。

```
def update(self):
    '''
    Please update self.W1 and self.W2
    '''

    # ==========
    # todo '''更新该前馈神经网络的参数'''
    # ==========
```

首先看 ReLU 的反向传播，由于 ReLU 的公式为（符号和课件中保持一致所以用了 a 和 x）

$$a = \begin{cases} x, & x > 0 \\ 0, & x \leqslant 0 \end{cases}$$

所以显然

$$\frac{\mathrm{d}a}{\mathrm{d}x} = \begin{cases} 1, & x > 0 \\ 0, & x \leqslant 0 \end{cases}$$

```python
class Relu:
    def __init__(self):
        self.mem = {}

    def forward(self, x):
        self.mem['x'] = x
        return np.where(x > 0, x,
        ↪  np.zeros_like(x))

    def backward(self, grad_y):
        '''
        grad_y: same shape as x
        '''

        # ==========
        # todo '''请完成激活函数的梯度后传'''
        # ==========
```

由于要计算梯度时要根据输入 x 是否大于 0 判断，所以这里使用了`self.mem` 来记忆上次输入的 x，在反向传播的时候就可以使用记忆的 x 来进行分支，这里可以利用 numpy 的批量操作能力实现，`grad_y` 是传入的梯度，返回的结果应为本层梯度与传入梯度的乘积：

$$return = \frac{\mathrm{d}a}{\mathrm{d}x} \times grad\_y = \begin{cases} grad\_y, & x > 0 \\ 0, & x \leqslant 0 \end{cases}$$

因此写出代码如下：

```
return np.where(self.mem['x'] > 0, grad_y, np.zeros_like(grad_y))
```

```
class Relu:
    def __init__(self):
        self.mem = {}

    def forward(self, x):
        self.mem['x'] = x
        return np.where(x > 0, x,
        ↪  np.zeros_like(x))

    def backward(self, grad_y):
        '''
        grad_y: same shape as x
        '''

        # ==========
        # todo '''请完成激活函数的梯度后传'''
        return np.where(self.mem['x'] > 0,
        ↪  grad_y, np.zeros_like(grad_y))
        # ==========
```

交叉熵损失的函数为

$$loss = \sum_{\text{每个类别}i} -y_i \log(\hat{y}_i)$$

L2 正则项的损失为 $\lambda \|W\|$，λ 为系数，W 为权重，距离用的是欧几里得距离，即

$$\sqrt{\sum_{W\text{中的每个参数}x} x^2}$$

这里有两层网络，也就是两层权重，所以

$$L2 = \lambda_1 \|W_1\| + \lambda_2 \|W_2\|$$

```python
def compute_loss(self, log_prob, labels):
    '''
    log_prob is the predicted probabilities
    labels is the ground truth
    Please return the loss
    '''

    # ==========
    # todo '''请完成多分类问题的损失计算 损失为：交叉熵损失 + L2 正则项'''
    # ==========
```

`log_prob` 应该是希望传入已经经过 $\log$ 计算的 $\hat{y}$，但是在 lab4.ipynb 里发现其实是没有经过 $\log$ 计算的 `pred_y`，这里还得自己计算 $\log(\hat{y})$，但是 $\log(\hat{y}_i)$ 由于在前向传播的时候计算过就提前缓存在`self.log_value` 了。

`labels`$|y$ 和`self.log_value`$|\log(\hat{y})$ 是 one-hot 编码的，形状为 [批大小，类别数]，根据公式在类别数维度求和，所以是 `axis=1`。注意还要在批大小维度求平均，即`.mean(0)`。

计算距离这里直接使用了 `np.linalg.norm`。

```python
def compute_loss(self, log_prob, labels):
    '''
    log_prob is the predicted probabilities
    labels is the ground truth
    Please return the loss
    '''

    # ==========
    # todo '''请完成多分类问题的损失计算 损失
    ↪  为: 交叉熵损失 + L2 正则项'''
    return - np.sum(labels * self.log_value,
    ↪  axis=1).mean(0) + self.lambda1 *
    ↪  np.linalg.norm(self.W1) +
    ↪  self.lambda1 *
    ↪  np.linalg.norm(self.W2)
    # ==========
```

这里 x 的形状是 [批大小，28，28]，这里的两个 28 分别是图像高度和宽度，而且可以观察到`self.W1` 的形状是 [100, 785]，但是 $28 \times 28 = 784$，说明需要把高度和宽度拉平后还需要拼接一个 `np.ones` 来替代偏置项的作用。即

```
np.concatenate((x.reshape(x.shape[0], -1),
↪  np.ones((x.shape[0], 1))), axis=1)
```

在 `Matmul.backward` 的注释中可以看到 `x: shape(d, N)`，所以拼接好之后还需要进行转置。

```
def forward(self, x):
    '''
    x is the input features
    Please return the predicted
    ↪  probabilities of x
    '''

    # ==========
    # todo '''请搭建一个 MLP 前馈神经网络 补
    ↪  全它的前向传播 MLP 结构为 FFN -->
    ↪  RELU --> FFN --> Softmax'''
    # ==========
```

在 `Softmax.forward` 的注释中可以看到 `x: shape(N, c)`，因此在进行 Softmax 操作前还需要再转置回来。

理论上这时候就可以直接返回了，不需要用到`self.log`，log 是在计算交叉熵时才会用到的操作，但是在 lab4.ipynb 中非要先反向传播再计算损失，反向传播需要`self.log.backward`，但这又需要先调用过`self.log.forward` 才能把输入记忆到`self.mem` 中，才能正确返回梯度。

那没办法，只能先调用一下`self.log.forward` 把结果缓存起来。

```python
def forward(self, x):
    '''
    x is the input features
    Please return the predicted
    ↪  probabilities of x
    '''

    # ==========
    # todo '''请搭建一个 MLP 前馈神经网络 补
    ↪  全它的前向传播 MLP 结构为 FFN -->
    ↪  RELU --> FFN --> Softmax'''
    y =
    ↪  np.concatenate((x.reshape(x.shape[0],
    ↪  -1), np.ones((x.shape[0], 1))),
    ↪  axis=1).T  # 这形状真难弄
    y = self.mul_h1.forward(self.W1, y)
    y = self.relu.forward(y)
    y = self.mul_h2.forward(self.W2, y).T
    y = self.softmax.forward(y)
    # print(y)
    # 唉没办法，非要先反向传播再计算损失，那只
    ↪  能把 log 的结果缓存起来了
    self.log_value = self.log.forward(y)
    return y
    # ==========
```

前面的准备工作都实现了后，这里就很简单了，只需要逐层反向传播就行了。

注意交叉熵损失为

$$loss = \sum_{\text{每个类别}i} -y_i \log(\hat{y}_i)$$

所以

$$\frac{\mathrm{d}loss}{\mathrm{d}\log(\hat{y}_i)} = -y_i$$

因此首个梯度为 `-label`，后续的反向传播就交给各层的 `backward` 函数了。

```python
def backward(self, label):
    '''
    label is the ground truth
    Please compute the gradients of self.W1
    ↪  and self.W2
    '''

    # ==========
    # todo '''补全该前馈神经网络的后向传播算
    ↪  法'''
    # ==========
```

仍然要注意在 Softmax 反向传播后需要转置一下。

`Matmul.backward` 返回的结果为`return grad_x, grad_W`，这也提示了全连接层要保留对输入和对参数的求导，对输入的求导用来继续反向传播，对参数的求导用来更新参数。

```
def backward(self, label):
    '''
    label is the ground truth
    Please compute the gradients of self.W1
    ↪  and self.W2
    '''

    # ==========
    # todo '''补全该前馈神经网络的后向传播算
    ↪  法'''
    temp = self.log.backward(-label)
    temp = self.softmax.backward(temp).T
    temp, self.gradient2 =
    ↪  self.mul_h2.backward(temp)
    temp = self.relu.backward(temp)
    temp, self.gradient1 =
    ↪  self.mul_h1.backward(temp)
    # ==========
```

更新参数只需要按照公式即可，不要忘记 L2 正则项的梯度，以下以 W_1 为例，W_2 同理。

$W_1^{(i,j)}$ 表示 W_1 的第 i 行 j 列的元素，lr 表示 learning rate，即学习率。

$$\frac{\mathrm{d}L2}{\mathrm{d}W_1^{(i,j)}} = \frac{2\lambda_1 W_1^{(i,j)}}{\|W_1\|}$$

$$W_1 = W_1 - \left(\frac{\mathrm{d}loss}{\mathrm{d}W_1} + \frac{\mathrm{d}L2}{\mathrm{d}W_1}\right) \times lr$$

```python
def update(self):
    '''
    Please update self.W1 and self.W2
    '''

    # ==========
    # todo '''更新该前馈神经网络的参数'''
    self.W1 -= (self.gradient1 + 2 *
    ↪  self.lambda1 * self.W1 /
    ↪  np.linalg.norm(self.W1)) * self.lr
    self.W2 -= (self.gradient2 + 2 *
    ↪  self.lambda1 * self.W2 /
    ↪  np.linalg.norm(self.W2)) * self.lr
    # ==========
```

感谢观看!