

1 Multi-Layer Fully Connected Neural Networks

在之前的作业中，我们实现了一个两层的神经网络。但是这个神经网络并没有模块化。而assignment2将前向传播、反向传播等模块化，这就使得我们可以通过这些代码搭建出任意的神经网络。

在Multi-Layer Fully Connected Neural Networks中，需要完成FullyConnectedNets.ipynb和layers.py相关代码。

1.1 affine_forward

```
x = x.reshape(x.shape[0], -1)
out = x.dot(w) + b
```

这里的代码就很简单了，不过要注意要把x的维度问题。

1.2 affine_backward

```
def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ..., d_k)
      - w: Weights, of shape (D, M)
      - b: Biases, of shape (M,)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    #####
    # TODO: Implement the affine backward pass. #
    #####
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x = x.reshape(x.shape[0], -1)
    dw = x.T.dot(dout)
    db = np.sum(dout, axis = 0)
    dx = dout.dot(w.T)
    dx = dx.reshape(x.shape)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #####
```

```
#                                     END OF YOUR CODE                                     #
#####
return dx, dw, db
```

有了前面的基础，我觉得这部分看起来就简单多了。当然，还是要注意输入x的维度。第一步就是把x拉直，reshape成 (x.shape[0], -1)。 $out = Xw + b$ ，所以对w求梯度就是 $\frac{\partial out}{\partial w} = X^T$ ，那么根据链式法则， $dw = X^T @ dout$ 。同样的。 $db = np.sum(dout, axis = 0)$ ， $dx = dout.dot(w.T)$ 。最后把dx的维度变回之前的维度即可。

1.3 relu_forward

```
out = np.maximum(x, 0)
```

1.4 relu_backward

```
def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLU).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    dx, x = None, cache
    #####
    # TODO: Implement the ReLU backward pass.                                     #
    #####
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    dx = (x > 0) * dout

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #####
    #                                     END OF YOUR CODE                                     #
    #####
    return dx
```

relu函数只有在大于0时才有梯度，且梯度为1。

1.5 affine_relu_forward

```
def affine_relu_forward(x, w, b):
    """
    Convenience layer that performs an affine transform followed by a ReLU

    Inputs:
    - x: Input to the affine layer
    - w, b: Weights for the affine layer
```

```

Returns a tuple of:
- out: Output from the ReLU
- cache: Object to give to the backward pass
"""
a, fc_cache = affine_forward(x, w, b)
out, relu_cache = relu_forward(a)
cache = (fc_cache, relu_cache)
return out, cache

```

现在开始就要利用之前所搭建的模块去拼接一个更大的模块了。这个函数完成了 $affine \rightarrow relu$ 的过程。所以首先调用前向传播函数，在调用relu函数。这里使用到了两个缓存，fc_cache和relu_cache。

```

affine_forward: cache = (x, w, b)
relu_forward: cache = x

```

在原函数中，两个缓存存储的结果如上述所示，有的朋友可能对此不太理解，但是看到反向传播的时候应该就会明白了。

1.6 affine_relu_backward

```

def affine_relu_backward(dout, cache):
    """
    Backward pass for the affine-relu convenience layer
    """
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db

```

这个是1.5函数反向传播的代码。实际上，上面的缓存正是为了向之前的反向传播传递参数用的。这里可以在仔细研究一下这几个函数的关系和构造方式，这对之后代码的书写很有帮助。

1.7 Softmax and SVM

这里建议复习一下这两个函数的反向传播过程。

1.8 TwoLayerNet

1.8.1 __init__

```

self.params['w1'] = weight_scale * np.random.randn(input_dim, hidden_dim)
self.params['b1'] = np.zeros(hidden_dim)
self.params['w2'] = weight_scale * np.random.randn(hidden_dim,
num_classes)
self.params['b2'] = np.zeros(num_classes)

```

1.8.2 loss

```

# 第一部分
layer1_out, layer1_cache = affine_relu_forward(X, self.params['w1'],
self.params['b1'])

```

```

        scores, layer2_cache = affine_forward(layer1_out, self.params['w2'],
self.params['b2'])
        # 第二部分
        loss, dscores = softmax_loss(scores, y)
        loss += 0.5 * self.reg * (np.sum(self.params['w1'] ** 2) +
                                np.sum(self.params['w2'] ** 2))
        dlayer1_out, dw2, db2 = affine_backward(dscores, layer2_cache)
        grads['w2'] = dw2 + self.reg * self.params['w2']
        grads['b2'] = db2

        dx, dw1, db1 = affine_relu_backward(dlayer1_out, layer1_cache)
        grads['w1'] = dw1 + self.reg * self.params['w1']
        grads['b1'] = db1

```

这里就像搭积木一样，把各个函数用好就行，相信大家看到代码是都能够理解的。

1.9 Solver

这里需要对这个cs231n/solver.py的API有一个明确的了解，这里只简单列举一下比较重要的几个API的作用。

成员函数名	参数	作用
<code>__init__</code>	建议仔细阅读源代码中的文档字符串，这里的参数包含但不只有选用的模型、数据、优化策略、学习率等	初始化类，给予模型必要的参数和数据集
<code>_step</code>	无	执行选用的优化策略对参数进行梯度更新
<code>train</code>	无	对模型进行训练
<code>check_accuracy</code>	X:测试数据集 y:测试数据集所对应的标签 num_samples:可输入一个整数作为参数，输入后表示选择测试集中的num_samples个数据进行精确度的验证，默认值为None batch_size:用于优化内存的参数，默认值为100	获得输入数据集的精确度

```

data = {
    'x_train': # training data
    'y_train': # training labels
    'x_val': # validation data
    'y_val': # validation labels
}
model = MyAwesomeModel(hidden_size=100, reg=10)
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={

```

```

        'learning_rate': 1e-3,
    },
    lr_decay=0.95,
    num_epochs=10, batch_size=100,
    print_every=100)

solver.train()

```

这个是官方给的示例。data是数据集，包括训练数据集和测试数据集；model是训练的模型。在Solver中需要传入model，data，update_rule等参数，最后调用solver.train函数对模型训练即可得到训练好的模型。基于此，我们可以在jupyter notebook中的代码。

```

solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                    'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=5, batch_size=100,
                print_every=100)

solver.train()
acc = solver.check_accuracy(data['X_test'], data['y_test'])
print("test acc:", acc)

```

1.10 Multilayer network

这里需要完成的是cs231n/classifiers/fc_net.py。从现在开始，我们要真正开始搭建积木了。

1.10.1 __init__

```

pre_dim = input_dim
for i, hidden_dim in enumerate(hidden_dims):
    rank = str(i + 1)
    self.params['W' + rank] = weight_scale * np.random.randn(pre_dim,
hidden_dim)
    self.params['b' + rank] = np.zeros(hidden_dim)
    pre_dim = hidden_dim
    self.params['W' + str(self.num_layers)] = weight_scale *
np.random.randn(pre_dim, num_classes)
    self.params['b' + str(self.num_layers)] = np.zeros(num_classes)

if self.normalization:
    for i in range(1, self.num_layers):
        rank = str(i)
        self.params['gamma' + rank] = np.ones(hidden_dims[i - 1])
        self.params['beta' + rank] = np.zeros(hidden_dims[i - 1])

```

在构造函数中仍然有很多参数，这里还是建议去阅读文档字符串。

构造函数是要实现全连接神经网络的参数初始化。而全连接神经网络的结构就是：

$$X \rightarrow XW_1 + b_1(H_1) \rightarrow H_1W_2 + b_2(H_2) \rightarrow H_2W_3 + b_3(H_3) \rightarrow \dots \rightarrow H_{n-1}W_n + b_n(Out)$$

所以对于输入的 H_i ，其维度是 (N, D_i) ，那么 W_{i+1} 的第一个维度就是 D_i ，第二个维度由输入的参数hidden_dims所决定。而在神经网络中，可能会使用normalization，所以还需要对gamma和beta进行初始化。这里都是按照要求初始化为0或1的。

1.10.2 loss

建议先阅读第二部分的BatchNormalization和第三部分Dropout。

1.10.2.1 四个函数

我们先来写四个辅助函数从而简化含有normalization步骤的神经网络传播代码。

- 函数一

```
def affine_bn_relu_forward(x, w, b, gamma, beta, bn_param):
    a, fc_cache = affine_forward(x, w, b)
    bn, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
    out, relu_cache = relu_forward(bn)
    cache = (fc_cache, bn_cache, relu_cache)
    return out, cache
```

这个函数用于 $affine_forward \rightarrow BN \rightarrow relu_forward$ 层。

这里都是很简单的函数调用，但是这三者的返回值，也就是三个cache要保留，并合并成一个cache，这将用于该层的反向传播代码中。

- 函数二

```
def affine_bn_relu_backward(dout, cache):
    fc_cache, bn_cache, relu_cache = cache
    dbn = relu_backward(dout, relu_cache)
    da, dgamma, dbeta = batchnorm_backward_alt(dbn, bn_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db, dgamma, dbeta
```

可以看到，我们利用之前记录的三个cache值，非常简单地就求出了反向传播的梯度。

- 函数三

```
def affine_ln_relu_forward(x, w, b, gamma, beta, ln_param):
    af_out, af_cache = affine_forward(x, w, b)
    ln_out, ln_cache = layernorm_forward(af_out, gamma, beta, ln_param)
    relu_out, relu_cache = relu_forward(ln_out)
    cache = (af_cache, ln_cache, relu_cache)
    return relu_out, cache
```

- 函数四

```
def affine_ln_relu_backward(dout, cache):
    af_cache, ln_cache, relu_cache = cache
    drelu = relu_backward(dout, relu_cache)
    dln, dgamma, dbeta = layernorm_backward(drelu, ln_cache)
    dx, dw, db = affine_backward(dln, af_cache)
    return dx, dw, db, dgamma, dbeta
```

1.10.2.2 forward

```
layer_input = x
caches = []
if self.normalization is None:
    for i in range(self.num_layers - 1):
        cache = []
        layer_input, all_cache = affine_relu_forward(layer_input,
            self.params['w%s' % (i + 1)], self.params['b%s' % (i + 1)])
        cache.append(all_cache)
        if self.use_dropout:
            layer_input, dt_cache = dropout_forward(layer_input,
                self.dropout_param)
            cache.append(dt_cache)
        caches.append(cache)
    else:
        ...
```

我们先来看不需要normalization的升降网络是如何前向传播的。

很显然，第一层输入就是X，caches用来储存前向传播过程中的返回值cache，这些返回值是用于反向传播中的。在assignment2中，把前向传播的返回值cache作为反向传播的参数就可求出梯度。这里要注意，我把affine_relu_forward层的所有cache和dropout层（如果使用dropout）的dt_cache都放入了cache中，再把cache放入了caches中。也就是说，如果训练时使用了dropout，那么反向传播的参数cache应该是caches[i][-1]。

```
else:
    if self.normalization == 'batchnorm':
        forward_func = affine_bn_relu_forward
    else:
        forward_func = affine_ln_relu_forward
    for i in range(self.num_layers - 1):
        cache = []
        layer_input, all_cache = forward_func(layer_input,
            self.params['w%s' % (i + 1)],
            self.params['b%s' % (i + 1)],
            self.params['gamma%s' % (i + 1)],
            self.params['beta%s' % (i + 1)],
            self.bn_params[i])
        cache.append(all_cache)
        if self.use_dropout:
            layer_input, dt_cache = dropout_forward(layer_input,
                self.dropout_param)
            cache.append(dt_cache)
        caches.append(cache)
```

现在我们再来看一看需要normalization时是如何进行前向传播的。

因为normalization的方式有两种，一种是BN，一种是LN，这要根据参数选择。现在需要用到我们之前写的四个辅助函数了。为简化代码，可以把affine_forward，BN/LN，relu_forward合并成一个函数：affine_bn_relu_forward/affine_ln_relu_forward。我们用forward_func根据参数选择合适的函数，再把对应的参数送入forward_func中，前向传播就完成了。当然，还有dropout，这里的处理方法和无normalization是一致的。

```

        scores, fc_cache = affine_forward(layer_input, self.params['w%s' %
self.num_layers],
                                          self.params['b%s' % self.num_layers])
        caches.append(fc_cache)

```

这是最后一次前向传播。至此，前向传播大功告成！

1.10.2.3 loss

```

loss, dout = softmax_loss(scores, y)
# 加上正则项
for i in range(1, self.num_layers + 1):
    loss += 0.5 * self.reg * np.sum(self.params['w%s' % i] ** 2)

```

1.10.2.4 backward

```

dout, dw, db = affine_backward(dout, caches[self.num_layers - 1])
grads['w%s' % self.num_layers] = dw + self.reg * self.params['w%s' %
self.num_layers]
grads['b%s' % self.num_layers] = db

```

我们先求出最后一次前向传播的梯度。

```

if self.normalization is None:
    for i in range(self.num_layers - 1, 0, -1):
        if self.use_dropout:
            dout = dropout_backward(dout, caches[i - 1][1])
            dout, dw, db = affine_relu_backward(dout, caches[i - 1][0])
            grads['w%s' % i] = dw + self.reg * self.params['w%s' % i]
            grads['b%s' % i] = db
        else:
            ...

```

同样的，如果不存在normalization，直接利用affine_relu_backward函数进行反向传播代码即可。注意：在前向传播中，caches储存的结果是这样的[[cache], [cache], [cahce]...]或是[[cache, dt_cache], [cache, dt_cache], [cahce, dt_cache]...]。也就是说，caches中储存的是一个一个的列表，所以还需要再将列表中的元素取出才能作为参数放入函数中。

```

else:
    if self.normalization == 'batchnorm':
        backward_func = affine_bn_relu_backward
    elif self.normalization == 'layernorm':
        backward_func = affine_ln_relu_backward
    for i in range(self.num_layers - 1, 0, -1):
        if self.use_dropout:
            dout = dropout_backward(dout, caches[i - 1][-1])
            dout, dw, db, dgamma, dbeta = backward_func(dout, caches[i - 1]
[0])

            grads['gamma%s' % i] = dgamma
            grads['beta%s' % i] = dbeta
            grads['w%s' % i] = dw + self.reg * self.params['w%s' % i]
            grads['b%s' % i] = db

```


带有normalization操作的反向传播也是类似的。

1.11 梯度更新

1.11.1 SGD+Momentum

```
v = config['momentum'] * v - config['learning_rate'] * dw
next_w = w + v
```

1.11.2 Adam

```
config['t'] += 1
beta1 = config['beta1']
beta2 = config['beta2']
epsilon = config['epsilon']
learning_rate = config['learning_rate']
config['m'] = beta1 * config['m'] + (1-beta1) * dw
config['v'] = beta2 * config['v'] + (1-beta2) * dw**2
mb = config['m'] / (1 - beta1**config['t'])
vb = config['v'] / (1 - beta2**config['t'])
next_w = w - learning_rate * mb / (np.sqrt(vb)+epsilon)
```

1.11.3 RMSProp

```
config['cache'] = config['decay_rate'] * config['cache'] + (1 -
config['decay_rate']) * (dw ** 2)
next_w = w - config['learning_rate'] * dw / np.sqrt(config['cache'] +
config['epsilon'])
```

1.12 Train a good model

```
learning_rate = 2.5e-2
weight_scale = 4e-2
reg = 1e-2
model = FullyConnectedNet([120, 120, 120, 120], normalization="batchnorm", reg =
1e-2,
                        weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, data,
                print_every=500, num_epochs=20, batch_size=100,
                update_rule='sgd',
                optim_config={
                    'learning_rate': learning_rate,
                })
solver.train()
```

这里我没有仔细调参，所以准确率也只有50%~60%左右。

2 BatchNormalization

虽然这里叫做BatchNormalization，但是这里实际上还包含了LayerNormalization的部分。在写代码之前，我觉得有必要将这两篇论文仔细阅读一番的。

2.1 Batch Normalization论文简单分析

2.1.1 问题引出

我们在运行assignment1的时候想必已经发现了，程序在有的地方跑得是非常慢的。而我们训练的只是两层的神经网络，如果训练的是一个非常深的神经网络，训练速度可想而知。如果要加速训练速度，可以把learning_rates变大，但是这在一定程度上会导致accuracy降低。这篇Paper提出了一个非常棒的方法可以加快训练速度，BatchNormalization。当然，BatchNormalization不只是具有加快训练速度的优点，它还可以减少过拟合，减少Internal Covariate Shift等优点。Internal Covariate Shift是Paper中提出的一篇概念，大致意思就是输入数据的分布会随着传播在后面每一层中发生变化。神经网络的本质就是学习数据的分布规律。一旦数据分布发生变化，神经网络就会对参数进行更新以适应新的数据，这样就拖慢了网络的训练速度。以第二层为例，在训练过程中第一层的参数是不断变化的，所以输出到第二层数据的分布也是不断变化的。这篇Paper所要解决的实质问题就是这个Internal Covariate Shift。

2.1.2 BN算法

为了降低Internal Covariate Shift，作者提出了BN算法。可以这样理解BN算法：通常的神经网络是 $affine_forward \rightarrow activation\ function \rightarrow affine_forward \rightarrow activation\ function \dots$ 。但是，作者在这里加入了一个BN层。所以就变成了

$affine_forward \rightarrow BN \rightarrow activation\ function \rightarrow affine_forward \rightarrow BN \rightarrow activation\ function \dots$ 。

现在我们看一下BN算法的具体做法。

Paper中给出了一个归一化公式：

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

这个公式使得数据均值变为0，方差变为1。但是这种变化会破坏当前层网络对数据特征的学习。因此，作者引入了一个两个学习参数： γ, β 。对归一化的数据进行了线性变换：

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}.$$

因此，向前传播的数据就变成了：

$$\begin{aligned} \mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i && // \text{ mini-batch mean} \\ \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 && // \text{ mini-batch variance} \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} && // \text{ normalize} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) && // \text{ scale and shift} \end{aligned}$$

γ, β 这两个参数是用于恢复原数据的一部分信息的。当 $\gamma^k = \sqrt{\text{Var}[x^{(k)}]}$, $\beta^k = E[x^{(k)}]$ 时，此时就变成了原输入数据。当不等于这两个值时，会舍弃原输入数据的一部分信息，这有点类似于PCA降维。我个人认为BN算法之所以有效，就是舍弃了一部分噪声，使得输入分布更加规整。

2.1.3 反向传播

当然，我们知道了算法的流程，必须得通过反向传播计算梯度对参数进行更新。Paper中非常详细的给出了反向传播梯度计算公式：

$$\begin{aligned}\frac{\partial \ell}{\partial \hat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\ \frac{\partial \ell}{\partial \sigma_B^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2} \\ \frac{\partial \ell}{\partial \mu_B} &= \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_B)}{m} \\ \frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\partial \ell}{\partial \mu_B} \cdot \frac{1}{m} \\ \frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}\end{aligned}$$

在这些公式中并不涉及复杂的推导，都是链式法则的应用。要注意的是，为了防止方差为0的情况，在计算时加入了一个极小量：

$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}}$$

2.1.4 应用

我们知道，无论是均值还是方差，都是在训练网络的时候才会有的。但是如果网络已经训练好了，此时参数已经固定，我们采用 $E_\beta[\mu_\beta]$ 和 $\frac{m}{m-1} E_\beta[\sigma_\beta]$ 代替输入数据的均值和方差。这里是均值和方差的无偏估计。Paper中还提到了偏置项b。经过均值归一化后，这个偏置项b实际上会抵消掉，也就是说b参数是可以丢弃掉的。

2.2 BN forward

现在，根据论文中BN算法，我们来将其用代码实现。

```

# train
sample_mean = np.mean(x , axis=0)
sample_var = np.var(x, axis=0)
x_norm = (x - sample_mean) / np.sqrt(sample_var + eps)
out = gamma * x_norm + beta
cache = (x, sample_mean, sample_var, x_norm, gamma, beta, eps)
running_mean = momentum * running_mean + (1 - momentum) * sample_mean
running_var = momentum * running_var + (1 - momentum) * sample_var

# test
x_norm = (x - running_mean) / np.sqrt(running_var + eps)
out = gamma * x_norm + beta

```

现在看来BN前向传播就非常简单了。running_mean和running_var采用了一阶动量的思想，在test中是以这两个作为数据均值和方差的。这里cache储存的都是反向传播所需要的数据。

2.3 BN backward

```

m = dout.shape[0]
x, sample_mean, sample_var, x_norm, gamma, beta, eps = cache

```

我们先把cache里的值拿出来，再对Paper中所有提到偏导一个一个求。

2.3.1 \hat{x}

$$\frac{\partial \ell}{\partial \hat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

这个就很简单了，注意这个 \hat{x} 就是代码中的x_norm。所以代码即为：

```
dx_norm = dout * gamma
```

2.3.2 σ_{β}^2

$$\frac{\partial \ell}{\partial \sigma_B^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

这里我们需要注意 σ_{β}^2 和 x_norm 的关系

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

所以代码可以写为：

```
dsample_var = np.sum(dx_norm * (-0.5) * x_norm / (sample_var + eps), axis = 0)
```

2.3.3 μ_{β}

$$\frac{\partial \ell}{\partial \mu_{\beta}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\beta}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\beta}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_{\beta})}{m}$$

直接套公式：

```
dsample_mean = np.sum(-dx_norm / np.sqrt((sample_var + eps)), axis = 0) + \
    dsample_var * np.sum(-2.0 * (x - sample_mean), axis = 0) / m
```

2.3.4 x_i

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\beta}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\beta}^2} \cdot \frac{2(x_i - \mu_{\beta})}{m} + \frac{\partial \ell}{\partial \mu_{\beta}} \cdot \frac{1}{m}$$

```
dx = dx_norm / np.sqrt(sample_var + eps) + \
    dsample_var * 2.0 * (x - sample_mean) / m + \
    dsample_mean / m
```

2.3.5 γ

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i$$

```
dgamma = np.sum(dout * x_norm, axis = 0)
```

2.3.6 β

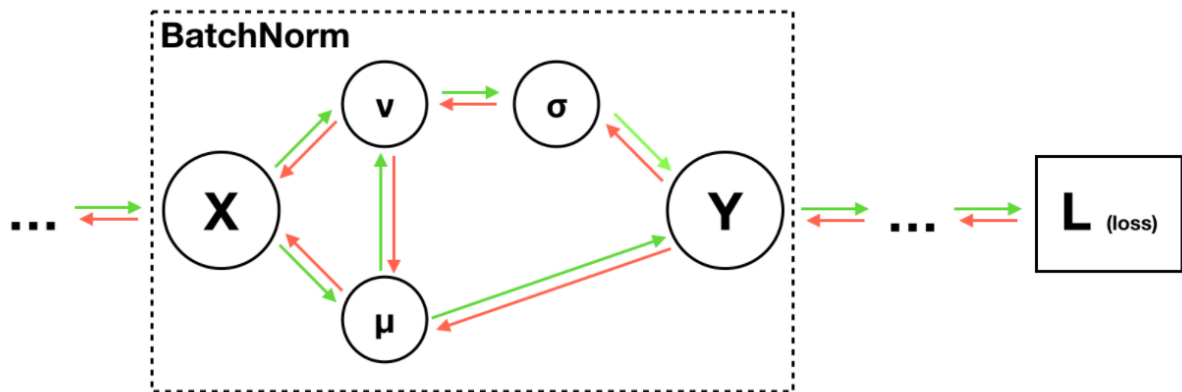
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

```
dbeta = np.sum(dout, axis = 0)
```

2.4 batchnorm_backward_alt

$$\mu = \frac{1}{N} \sum_{k=1}^N x_k \quad v = \frac{1}{N} \sum_{k=1}^N (x_k - \mu)^2$$

$$\sigma = \sqrt{v + \epsilon} \quad y_i = \frac{x_i - \mu}{\sigma}$$



作业中给出了计算图，但是我不太明白这里的意思，所以这部分代码和反向传播代码是一样的。

2.5 Layer Normalization

Layer Normalization和Batch Normalization很相似，但是进行操作的维度不一样。要注意的是，我们现在所用的网络并不是卷积神经网络，所以输入的数据只有两个维度。在Batch Normalization中我们是对Batch方向进行了Normalization操作，但是Layer Normalization将会对特征方向进行Normalization操作，也可理解为对输入数据的转置进行Batch Normalization操作。

2.6 LN forward

```
sample_mean = np.mean(x, axis=1, keepdims=True)
sample_var = np.var(x, axis=1, keepdims=True)
x_norm = (x - sample_mean) / np.sqrt(sample_var+eps)
out = x_norm * gamma + beta
cache = (x, sample_mean, sample_var, x_norm, gamma, beta, eps)
```

这里和BN是基本是一样的，区别在于操作的维度方向不同。

2.7 LN backward

```
D = dout.shape[1]
x, sample_mean, sample_var, x_norm, gamma, beta, eps = cache
dbeta = np.sum(dout, axis=0)
dgamma = np.sum(dout * x_norm, axis=0)

dx_norm = dout * gamma
dsample_var = np.sum(dx_norm * (-0.5) * x_norm / \
    (sample_var + eps), axis = 1, keepdims=True) # (N,1)
dsample_mean = np.sum(dx_norm * (-1) / np.sqrt(sample_var + eps), \
    axis = 1, keepdims=True) # (N,1)
dsample_mean += dsample_var * np.sum(-2 * (x - sample_mean), \
    axis = 1, keepdims=True) / D
dx = dx_norm / np.sqrt(sample_var + eps) + \
    dsample_var * 2 * (x - sample_mean) / D + \
    dsample_mean / D
```

LN的bp和BN的bp基本是一样的，只需将维度修改一下就好。要注意的是dgamma和dbeta，gamma和beta维度仍然是(D,)，并不是(N,)，所以梯度的公式和BN完全一致，不需要更改维度。

3 Dropout

3.1 Introduction

Dropout是在深度学习中防止过拟合的一大利器。它的思想非常简单，在每层的输入中忽略一定的神经元，也就是将其置零。我们通常设置一个概率p表示神经元置零的概率。在训练过程中，我们会使用dropout层，但是在预测时则不会使用。这是因为如果在预测时依旧随机舍弃一部分神经元，有可能会

3.2 forward

```
# train
mask = (np.random.rand(*x.shape) < p)
out = x * mask

# test
out = x
```

前向传播就很简单了。

3.3 backward

```
dx = dout * mask
```

显然只有mask中大于0的地方才存在梯度且为1，所以直接中dout和mask点乘就好。

4 CNN

4.1 forward

```
N, C, H, W = x.shape
F, C, HH, WW = w.shape
stride = conv_param['stride']
pad = conv_param['pad']
H_ap = 1 + (H + 2 * pad - HH) // stride
W_ap = 1 + (W + 2 * pad - WW) // stride
out = np.zeros((N, F, H_ap, W_ap))
x_padding = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode = \
    'constant', constant_values = 0)
for f in range(F):
    for i in range(H_ap):
        for j in range(W_ap):
            out[:, f, i, j] = np.sum(x_padding[:, :, \
                i * stride : i * stride + HH, j * stride : j * stride + WW ] \
                * w[f], axis = (1, 2, 3)) + b[f]
```

卷积的第一步应当是确定输出的维度。卷积核的维度是(F, C, HH, WW)，输入数据的维度是(N, C, H, W)，显然输出的维度应当是(N, F, xx, xx)。out的每一层的特征图的维度在代码提示里已经给出了计算公式。接下来只要利用循环把每张特征图计算出来就好。out[:, f, i, j]，代表的是输入数据的某个区域与第f个卷积核进行卷积运算所得到的结果，(i, j)对应的输入数据的范围就是x_padding[:, :, i * stride : i * stride + HH, j * stride : j * stride + WW]。

4.2 backward

```
x, w, b, conv_param = cache
stride, pad = conv_param['stride'], conv_param['pad']
N, C, H, W = x.shape
F, C, HH, WW = w.shape
H_ap = 1 + (H + 2 * pad - HH) // stride
W_ap = 1 + (W + 2 * pad - WW) // stride
x_padding = np.pad(x, ((0,0),(0,0),(pad,pad),(pad,pad)),
,mode='constant',constant_values=0)
dx_padding = np.zeros_like(x_padding)
dw = np.zeros_like(w)
db = np.zeros_like(b)
dx = np.zeros_like(x)
for f in range(F):
    for j in range(H_ap):
        for k in range(W_ap):
            db[f] += np.sum(dout[:,f,j,k], axis = 0)
            dw[f] += np.sum(x_padding[:, :, j*stride:j*stride+HH,
k*stride:k*stride+WW]*dout[:,f,j,k][:, None,None,None], axis = 0)
            dx_padding[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW] +=
w[f]*dout[:,f,j,k][:, None,None,None]

dx = dx_padding[:, :, pad:pad+H, pad:pad+W]
```

卷积的反向传播公式也不难，但是可能感觉上有点奇怪。对于out[n,f,i,j]来说，它是由x_padding[:,C,j*stride:j*stride+HH, k*stride:k*stride+WW]和卷积核w[f]进行卷积运算获得的。对于x_padding[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW]这个区域来说， $\frac{\partial out}{\partial x_padding} = w[f]$ 。当然求梯度还要乘dout。这里都是四维的矩阵乘法，有些复杂，建议在纸上画一画卷积的过程去感受一下。与求dx_padding类似，dw和db也比较简单。

4.3 max_pool_forward

```
N, C, H, W = x.shape
HH, WW, stride = pool_param['pool_height'], pool_param['pool_width'],
pool_param['stride']
H_ap = 1 + (H - HH) // stride
W_ap = 1 + (W - WW) // stride
out = np.zeros((N,C,H_ap,W_ap))
for j in range(H_ap):
    for k in range(W_ap):
        out[:, :, j, k] =
np.max(x[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW], axis=(2,3))
```

相比于卷积的抽象性，池化可能就抑郁理解多了，只需要在相应的区域取最大值就好。

4.4 max_pool_backward

```
x, pool_param = cache
N, C, H, W = x.shape
HH, WW, stride = pool_param['pool_height'], pool_param['pool_width'],
pool_param['stride']
H_ap = 1 + (H - HH) // stride
W_ap = 1 + (W - WW) // stride
dx = np.zeros_like(x)
for j in range(H_ap):
    for k in range(W_ap):
        square = x[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW]
        dx[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW] = (square ==
np.max(square, axis=(2, 3), keepdims=True)) * dout[:, :, j, k][:, :, None, None]
```

池化的反向传播也是有些绕。

`out[:, :, j, k] = np.max(x[:, :, j*stride:j*stride+HH, k*stride:k*stride+WW], axis=(2, 3))`，这说明最大池化层反向传播的结果是1,0。因此，我们把x这一块取出来命名为square，比较square中在第二、三维上的最大值，相等为1，不等为0。最后再乘以dout相应区间，注意维度问题即可。

4.5 说明

CNN部分代码分析的非常浅显，这是因为卷积是一个比较抽象的过程。如果真正想弄懂这些原理，看过程图是一个很好的方式。但是碍于绘画水平，只能用文字进行一些简单的描述。所以还是建议大家去找一些博客阅读，去真正理解这些过程的原理。

4.6 Spatial Batch Normalization

我们之前所进行的BN操作，是基于神经网络的。我们知道，神经网络的输入数据x维度是(N,D)。现在换成卷积神经网络，x的维度变成了(N,C,H,W)，那么我们如何进行BN操作呢？答案就是沿着channel方向，进行BN操作。

```
N, C, H, W = x.shape
new_x = x.transpose(0, 2, 3, 1).reshape(N * H * W, C)
out, cache = batchnorm_forward(new_x, gamma, beta, bn_param)
out = out.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

4.7 spatial_batchnorm_backward

```
N, C, H, W = dout.shape
new_dout = dout.transpose(0, 2, 3, 1).reshape(N * H * W, C)
dx, dgamma, dbeta = batchnorm_backward(new_dout, cache)
dx = dx.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

4.8 spatial_groupnorm_forward

```

N, C, H, W = x.shape
x_group = x.reshape((N, G, C//G, H, W))
x_mean = np.mean(x_group, axis=(2, 3, 4), keepdims=True)
x_var = np.var(x_group, axis=(2, 3, 4), keepdims=True)
x_groupnorm = (x_group - x_mean) / np.sqrt(x_var + eps)
x_norm = x_groupnorm.reshape((N, C, H, W))
out = x_norm * gamma + beta
cache = (G, x, x_norm, x_mean, x_var, beta, gamma, eps)

```

gn实际上就是把通道分组，在batch和组的方向上进行归一化，所以代码和BN十分相似。

4.9 spatial_groupnorm_backward

```

N, C, H, W = dout.shape
G, x, x_norm, x_mean, x_var, beta, gamma, eps = cache
dbeta = np.sum(dout, axis=(0, 2, 3), keepdims=True)
dgamma = np.sum(dout * x_norm, axis=(0, 2, 3), keepdims=True)

dx_norm = dout * gamma
dx_groupnorm = dx_norm.reshape((N, G, C // G, H, W))

x_group = x.reshape((N, G, C // G, H, W))
dx_var = np.sum(dx_groupnorm * -1.0 / 2 * (x_group - x_mean) / (x_var + eps)
** (3.0 / 2), axis=(2,3,4), keepdims=True)

m = C // G * H * W

dx_mean = np.sum(-dx_groupnorm / np.sqrt(x_var + eps), axis=(2,3,4),
keepdims=True) + \
    dx_var * -2.0 / m * np.sum(x_group - x_mean, axis=(2,3,4), keepdims=True)
dx_group = dx_groupnorm / np.sqrt(x_var + eps) \
    + dx_mean * 1.0 / m \
    + dx_var * 2.0 / m * (x_group - x_mean)


dx = dx_group.reshape((N, C, H, W))

```

可以发现，GN反向传播的代码和BN反向传播的代码也是一致的。唯一不同的在于m。

GN:

```
m = C // G * H * W
```

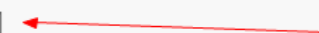


BN:

```

m = dout.shape[0]
x, sample_mean, sample_var, x_norm, gamma, beta, eps = cache

```



BN的m是dout的第一个维度，换成卷积就是N*H*W，而GN的m是C//G*H*W。这是二者不同的地方。

5 TensorFlow

留坑...

6 Pytorch

留坑...

7 Star

码字排版不易，给颗star吧...