Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

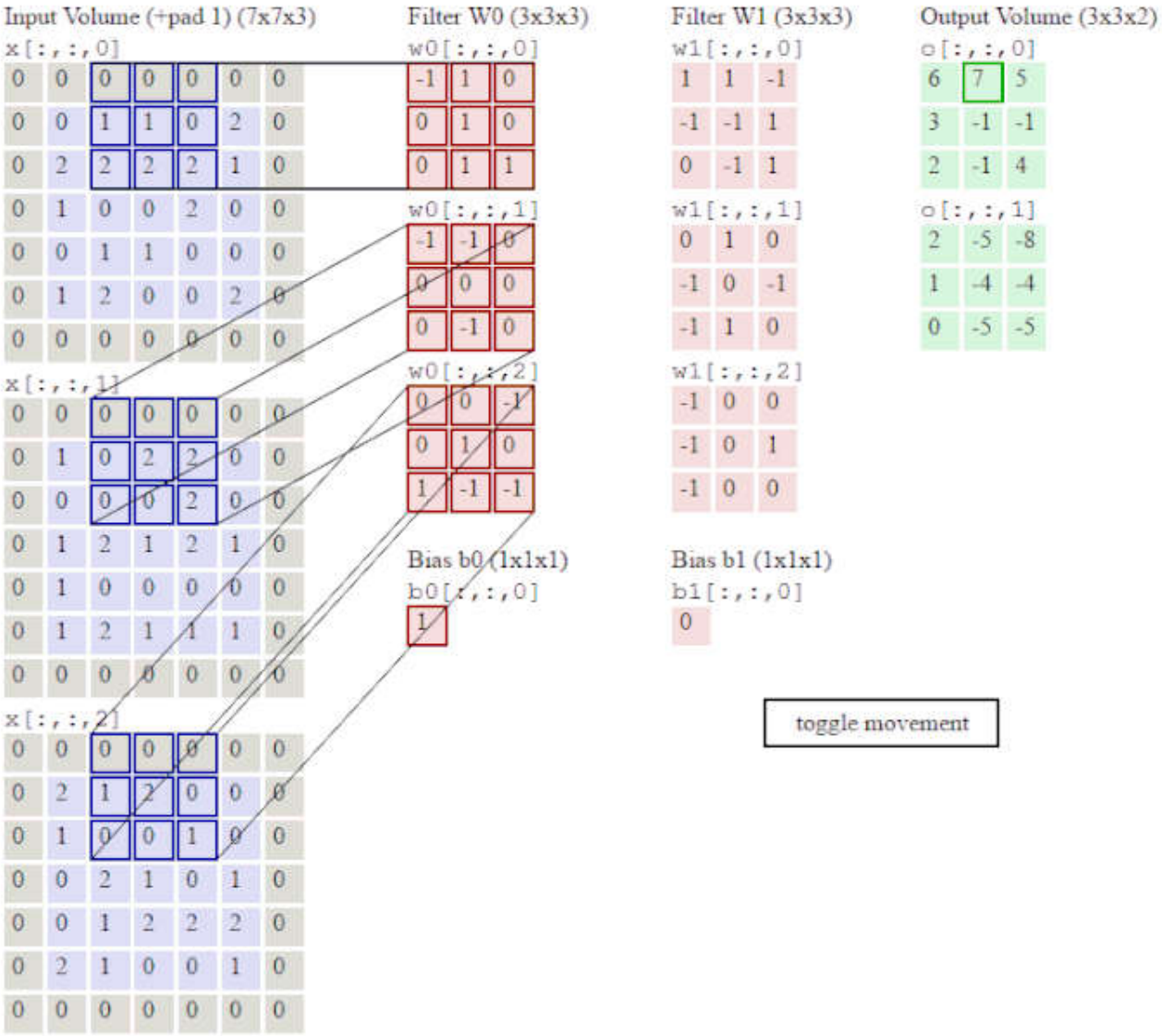
You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

注解：

卷积层简介：

卷积层，也可以称之为特征提取层，是CNNs最重要的部分。卷积层需要训练的参数是一系列的过滤器（也称卷积核），这些过滤器的大小一致，通常都是正方形。假设我们有n个过滤器，每个过滤器的大小为$k*k$（k通常取3或5），那么这一层我们需要训练的参数就有$n*k*k+n/c$个（这里的c表示通道数，如果是灰度图像c=1，如果是彩色图像c=3）。权值共享告诉我们，一个过滤器只能提取一种特征，即当过滤器在图像上卷积（滑动）的过程中，只提取了该图像全局范围内的同一个特征。所以，n个过滤器可以提取图像的n个不同特征。这里贴张卷积过程的动图来直观理解一下，这里的过滤器个数是6，但事实上是2种（因为有三个通道），所以提取了两种特征：

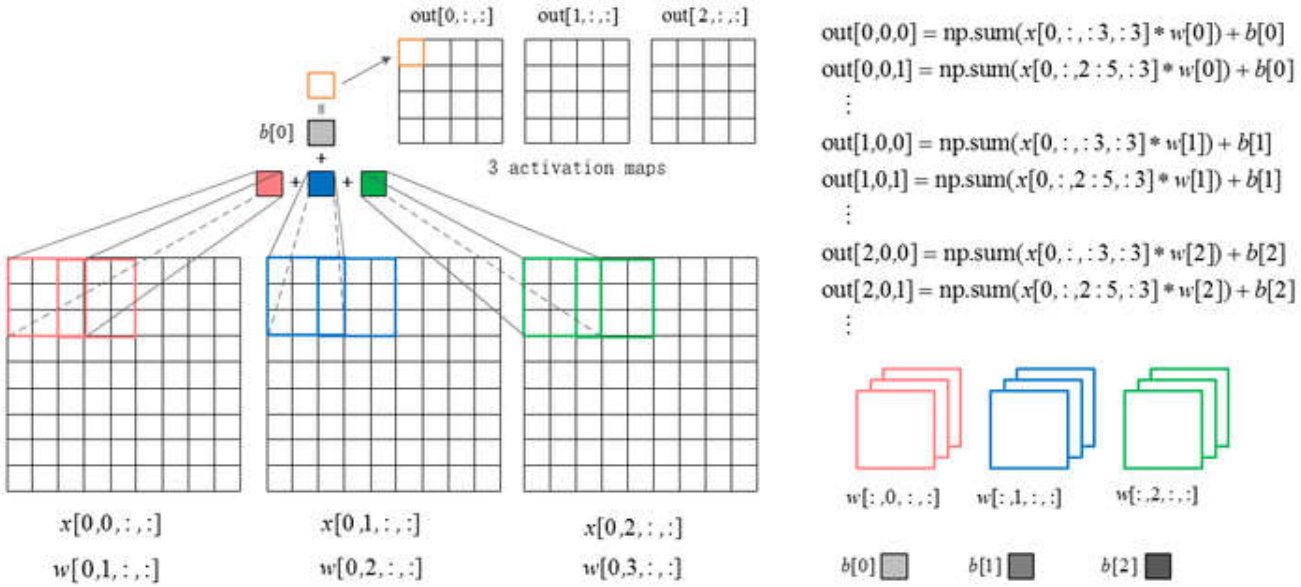


动图中，图像外面多了一圈0，而且过滤器移动的步长（stride）为2。补零这个操作，我们称之为zero-padding。我们记补零的圈数为p，过滤器移动步长为s，那么计算输出卷积特征（convolved feature，或者叫activation map）边长的公式为：$L = (input_d im - k + 2p)/s + 1$，输出特征的维数则为$L * L * n/c$。zero-padding这个操作产生的原因是为了保证过滤器的滑动能从头到尾刚刚好，即保证上面的公式能够整除。上面的p，s和n是需要我们提前设定好的三个超参数。对于步长s的设定，s设定得越小，提取的信息就越丰富，但计算量会相对大一点；s设定得越大，计算量会相对小一点，但是提取的信息就少一些。s的通常选择是1。

卷积为什么有效? 自然图像有其固有特性，也就是说，图像的一部分的统计特性与其他部分是一样的。这也意味着我们在这一部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。

卷积层的前向传播

我们通过以下这张图来直观说明卷积层的前向传播过程



我们假设:

1. 输入数据x 的shape为（N,C,H,W）,表示有N个样本，C个通道，图片的高和宽分别为H和W
2. 卷积核w的 shape(F,C,HH,WW) 表示有F个卷积核，C个通道，卷积核的高和宽分别为HH和WW
3. bias 的 shape 为（F，） 表示每个卷积核都有一个bias
4. 由上得到的输出out的shape位（N,F,HH',WW'）表示N个样本，对每个样本得到F个feature maps， 每个featrue map的 高和宽是 HH'和 WW'
   在图中的例子中N=1，C=3，F=3，卷积核移动步长 start_h=2。

   总结上图，可以得到前向传播公式：
   设，需要补零的圈数为pad，则：$HH' = 1 + (H + 2 * pad - HH)/stride$
   $WW' = 1 + (W + 2 * pad - WW)/stride$

   补零后的输入变为x_pad
   $out[:, i, j, k] = np.sum(x_pad[:, :, start_h : end_h, start_w : end_w] * w[i, :, :, :]) + b[i]$
   其中
   $start_h = min(j * stride, H + 2 * pad - HH) end_h = start_h + HH$
   $start_w = min(k * stride, W + 2 * pad - WW) end_w = start_w + WW$

代码如下：

```
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride = conv_param['stride']
    pad = conv_param['pad']
    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), 'constant')
    out_h = 1 + (H + 2 * pad - HH) // stride
    out_w = 1 + (W + 2 * pad - WW) // stride
    out = np.zeros([N, F, out_h, out_w])
    for j in range(out_h):
        for k in range(out_w):
            start_h = min(j * stride, H + 2 * pad - HH)
            end_h = start_h + HH
            start_w = min(k * stride, W + 2 * pad - WW)
            end_w = start_w + WW
            for i in range(F):
                out[:, i, j, k] = np.sum(x_pad[:, :, start_h:end_h, start_w:end_w] * w[i, :, :,
:], axis=(1, 2, 3))
    out = out + b[None, :, None, None]
```

```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

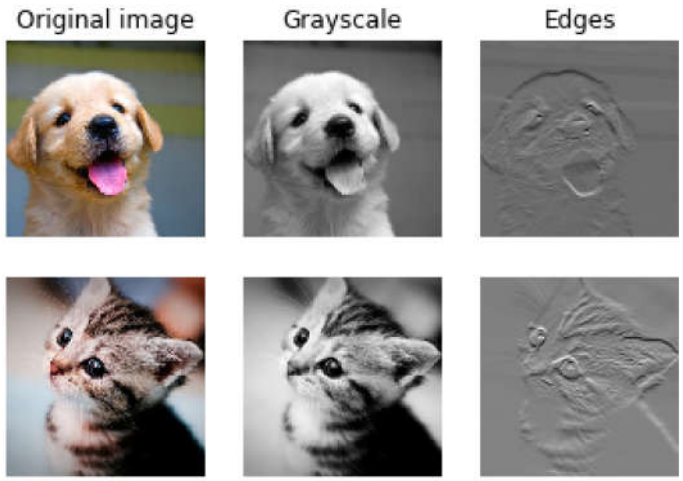
Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

```
from scipy.misc import imread, imresize

kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size)).transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
```

```
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
D:\Python\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: DeprecationWarning: `imread` is
deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
D:\Python\Anaconda3\lib\site-packages\ipykernel_launcher.py:10: DeprecationWarning: `imresize`
is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
D:\Python\Anaconda3\lib\site-packages\ipykernel_launcher.py:11: DeprecationWarning: `imresize`
is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```

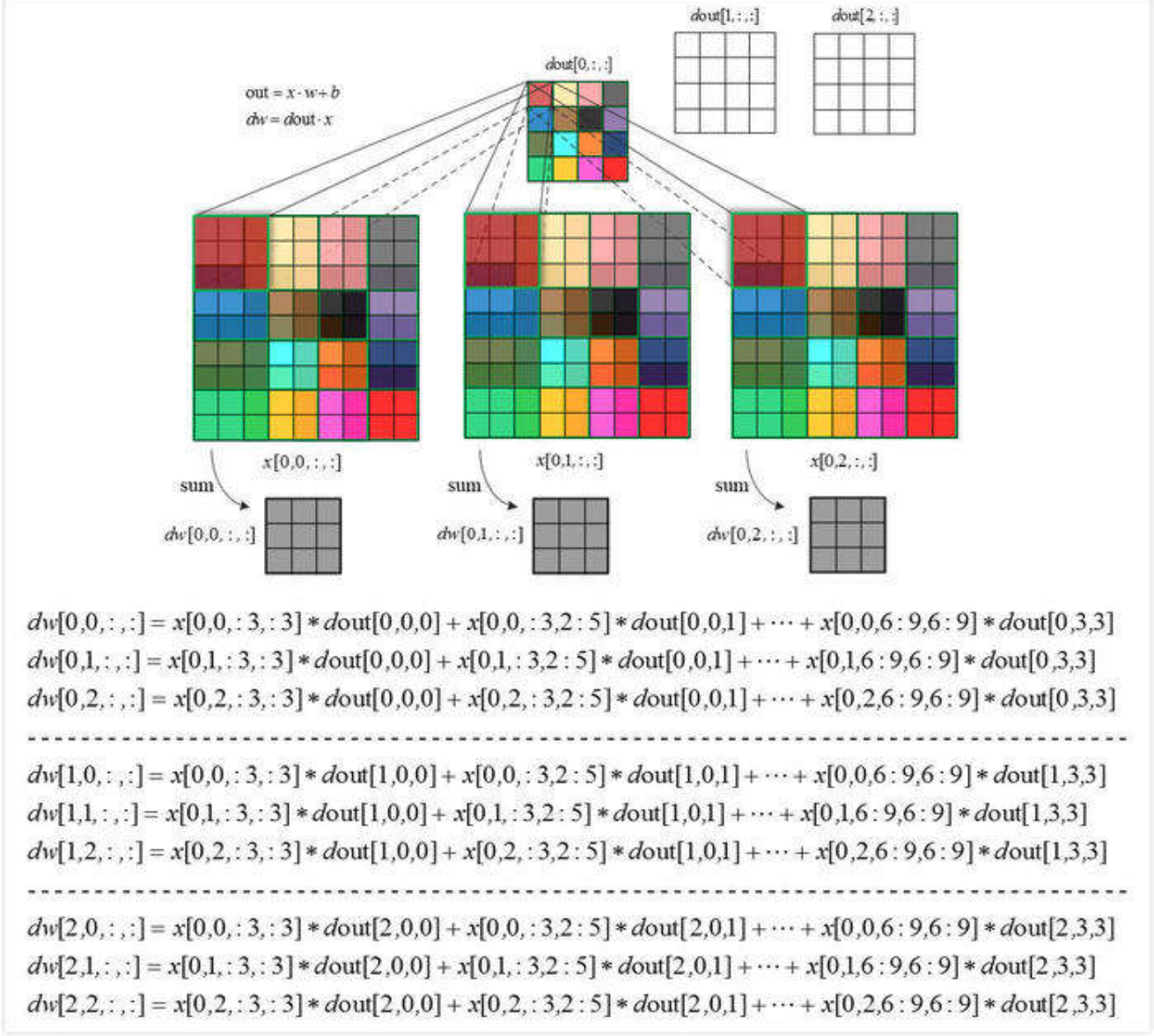


Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

注解



$$dw[0,0,:,:] = x[0,0,:3,:3] * dout[0,0,0] + x[0,0,:3,2:5] * dout[0,0,1] + \cdots + x[0,0,6:9,6:9] * dout[0,3,3]$$
$$dw[0,1,:,:] = x[0,1,:3,:3] * dout[0,0,0] + x[0,1,:3,2:5] * dout[0,0,1] + \cdots + x[0,1,6:9,6:9] * dout[0,3,3]$$
$$dw[0,2,:,:] = x[0,2,:3,:3] * dout[0,0,0] + x[0,2,:3,2:5] * dout[0,0,1] + \cdots + x[0,2,6:9,6:9] * dout[0,3,3]$$

$$dw[1,0,:,:] = x[0,0,:3,:3] * dout[1,0,0] + x[0,0,:3,2:5] * dout[1,0,1] + \cdots + x[0,0,6:9,6:9] * dout[1,3,3]$$
$$dw[1,1,:,:] = x[0,1,:3,:3] * dout[1,0,0] + x[0,1,:3,2:5] * dout[1,0,1] + \cdots + x[0,1,6:9,6:9] * dout[1,3,3]$$
$$dw[1,2,:,:] = x[0,2,:3,:3] * dout[1,0,0] + x[0,2,:3,2:5] * dout[1,0,1] + \cdots + x[0,2,6:9,6:9] * dout[1,3,3]$$

$$dw[2,0,:,:] = x[0,0,:3,:3] * dout[2,0,0] + x[0,0,:3,2:5] * dout[2,0,1] + \cdots + x[0,0,6:9,6:9] * dout[2,3,3]$$
$$dw[2,1,:,:] = x[0,1,:3,:3] * dout[2,0,0] + x[0,1,:3,2:5] * dout[2,0,1] + \cdots + x[0,1,6:9,6:9] * dout[2,3,3]$$
$$dw[2,2,:,:] = x[0,2,:3,:3] * dout[2,0,0] + x[0,2,:3,2:5] * dout[2,0,1] + \cdots + x[0,2,6:9,6:9] * dout[2,3,3]$$

总结上图，我们知道，dw的更新只和该卷积核涉及到的补零后的x相关，dx同理，其实就是 $\boldsymbol{out = x * w + b}$ 这个函数分别对x、w、b求导得到对应梯度，只不过x是一个感受野的范围。很容易得到下面代码，很直观了：

```
    x, w, b, conv_param = cache
    pad = conv_param['pad']
    stride = conv_param['stride']
    dx = np.zeros_like(x)
    dw = np.zeros_like(w)
    db = np.zeros_like(b)

    N, C, H,  W  = x.shape
    F, C, HH, WW = w.shape

    out_h = int(1 + (H + 2 * pad - HH) / stride)
    out_w = int(1 + (W + 2 * pad - WW) / stride)

    X  = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant')
    dX = np.pad(dx, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant')

    for i1 in range(N):
        for i2 in range(F):
            for i3 in range(out_h):
                for i4 in range(out_w):
                    start_h = i3 * stride
                    start_w = i4 * stride
                    end_h = start_h + HH
                    end_w = start_w + WW
                    dw[i2] += X[i1, :, start_h:end_h, start_w:end_w] * dout[i1, i2, i3, i4]
                    db[i2] += dout[i1, i2, i3, i4]
                    dX[i1, :, start_h:HH + start_h, start_w:WW + start_w] += w[i2] * dout[i1,
i2, i3, i4]

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

```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param)[0], x,
dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)[0], w,
dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param)[0], b,
dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around e-8 or less.
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

注解

Max-Pooling 层其实和卷积层类似，我们也可以假设有一个pooling 核（类比卷积核），pooling之后得到的feature map的高和宽的计算方法和前面卷积核的一样。和卷积层唯一的计算不同的是pooling层输入只有一个通道，然后pooling核不是框住的x的局部中，求这个局部的最大值（卷积层中这一步是求这个局部的和）。这样我们只需要稍微修改一下卷积层前向传播的代码就不难得到Max-Pooling的前向传播代码：

```
    N, C, H, W = x.shape
    ph, pw, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
    out_h = 1 + (H - ph) // stride
    out_w = 1 + (W - pw) // stride
    out = np.zeros([N, C, out_h, out_w])
    for i in range(out_h):
        for j in range(out_w):
            start_h = min(i * stride, H - ph)
            start_w = min(j * stride, W - pw)
            end_h = start_h + ph
            end_w = start_w + pw
            out[:, :, i, j] = np.max(x[:, :, start_h:end_h, start_w:end_w], axis=(2, 3))
```

```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be on the order of e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

注解

同样类似卷积层的方向传播，卷积层的反向传播只影响感受野的部分，而Max-Pooling的反向传播只影响感受野中的最大值部分。由于是单纯地求最大值，也就没有dw和db了。代码如下：

```
    x, pool_param = cache
    N, C, H, W = x.shape
    ph, pw, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
    out_h = 1 + (H - ph) // stride
    out_w = 1 + (W - pw) // stride
    dx = np.zeros_like(x)
    for i in range(out_h):
        for j in range(out_w):
            start_h = min(i * stride, H - ph)
            start_w = min(j * stride, W - pw)
            end_h = start_h + ph
            end_w = start_w + pw
            max_num = np.max(x[:, :, start_h:end_h, start_w:end_w], axis=(2,3))
            mask = (x[:, :, start_h:end_h, start_w:end_w] == (max_num)[:,:,None,None])
            dx[:, :, start_h:end_h, start_w:end_w] += (dout[:, :, i, j])[:,:,None,None] * mask
````


```python
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_param)[0], x,
dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be on the order of e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 0.190684s
Fast: 0.118486s
Speedup: 1.609337x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 7.399159s
Fast: 0.093566s
Speedup: 79.079619x
dx difference:  1.949764775345631e-11
dw difference:  5.188375174206562e-13
db difference:  3.481354613192702e-14
```

```
# Relative errors should be close to 0.0
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.010508s
fast: 0.004003s
speedup: 2.625112x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.025518s
fast: 0.015511s
speedup: 1.645147x
dx difference:  0.0
```

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, conv_param,
pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_param,
pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_param,
pool_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error:  5.828178746516271e-09
dw error:  8.443628091870788e-09
db error:  3.57960501324485e-10
```

```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_param)[0], x,
dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param)[0], w,
dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param)[0], b,
dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu:
dx error:  3.5600610115232832e-09
dw error:  2.2497700915729298e-10
db error:  1.3087619975802167e-10
```

Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

注解：

这里三层卷积网络的要求是1个卷积层（包含卷积层+relu+pooling）和2个全连接层。也就是在之前实现过的两层神经网络的基础上，在前面加卷积层就可以了。调用的接口是conv_relu_pool_forward 和 conv_relu_pool_backward。代码详见cnn.py

Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.302586071243987
Initial loss (with regularization):  2.508255638232932
```

Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
grads[param_name])))
```

```
W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

Overfit small data

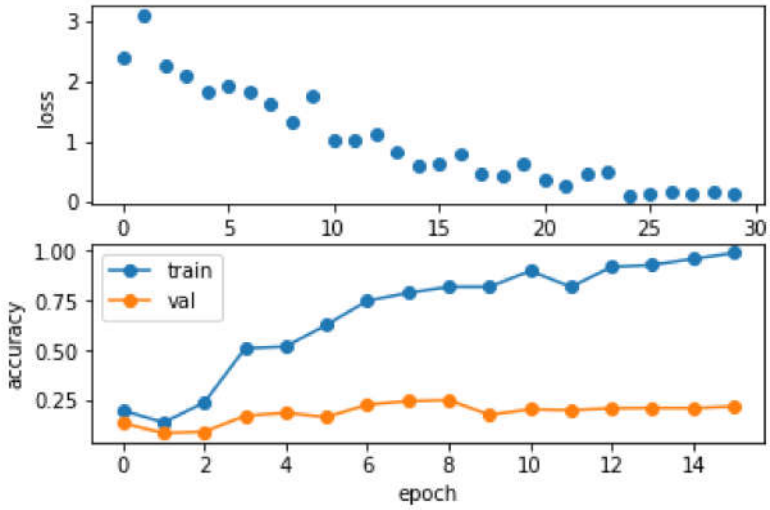
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
np.random.seed(231)

num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



注解

训练集过小，很容易过拟合训练集，但是验证集正确率非常低，因为训练集太小网络的泛化能力就很差了

Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

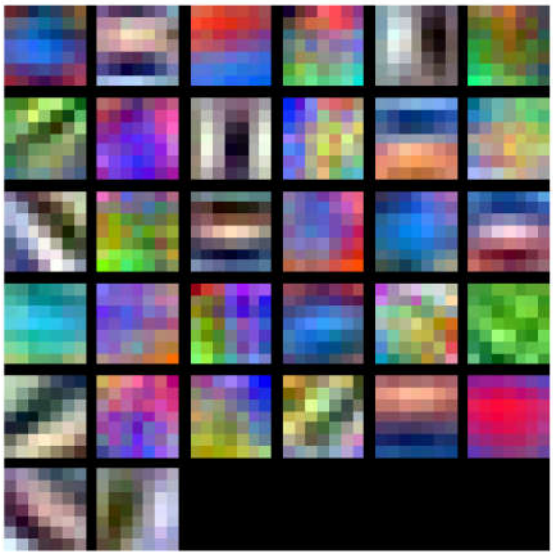
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304652
(Epoch 0 / 1) train acc: 0.095000; val_acc: 0.103000
(Iteration 21 / 980) loss: 2.188760
(Iteration 41 / 980) loss: 2.120930
(Iteration 61 / 980) loss: 2.272227
(Iteration 81 / 980) loss: 1.796872
(Iteration 101 / 980) loss: 1.849466
(Iteration 121 / 980) loss: 1.794737
(Iteration 141 / 980) loss: 1.697764
(Iteration 161 / 980) loss: 1.805761
(Iteration 181 / 980) loss: 1.768424
(Iteration 201 / 980) loss: 1.935073
(Iteration 221 / 980) loss: 1.891591
(Iteration 241 / 980) loss: 1.756048
(Iteration 261 / 980) loss: 2.066324
(Iteration 281 / 980) loss: 1.846859
(Iteration 301 / 980) loss: 1.646387
(Iteration 321 / 980) loss: 1.489619
(Iteration 341 / 980) loss: 1.651780
(Iteration 361 / 980) loss: 1.616642
(Iteration 381 / 980) loss: 1.757463
(Iteration 401 / 980) loss: 1.464414
(Iteration 421 / 980) loss: 1.586967
(Iteration 441 / 980) loss: 1.403613
(Iteration 461 / 980) loss: 1.473784
(Iteration 481 / 980) loss: 1.589312
(Iteration 501 / 980) loss: 1.953250
(Iteration 521 / 980) loss: 1.442303
(Iteration 541 / 980) loss: 1.450503
(Iteration 561 / 980) loss: 1.793000
(Iteration 581 / 980) loss: 1.630983
(Iteration 601 / 980) loss: 1.489644
(Iteration 621 / 980) loss: 1.357073
(Iteration 641 / 980) loss: 1.373433
(Iteration 661 / 980) loss: 1.524663
(Iteration 681 / 980) loss: 1.376444
(Iteration 701 / 980) loss: 1.707360
(Iteration 721 / 980) loss: 1.620943
(Iteration 741 / 980) loss: 1.686306
(Iteration 761 / 980) loss: 1.774393
(Iteration 781 / 980) loss: 1.729656
(Iteration 801 / 980) loss: 1.459732
(Iteration 821 / 980) loss: 1.489718
(Iteration 841 / 980) loss: 1.564707
(Iteration 861 / 980) loss: 1.659818
(Iteration 881 / 980) loss: 1.559641
(Iteration 901 / 980) loss: 1.480699
(Iteration 921 / 980) loss: 1.454675
(Iteration 941 / 980) loss: 1.404195
(Iteration 961 / 980) loss: 1.653157
(Epoch 1 / 1) train acc: 0.452000; val_acc: 0.453000
```

Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different imagesand different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

[3] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

Spatial batch normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

注解

batch normalization 是全连接层用的，输入和输出都是（N,D）的矩阵。这里卷积层也要用BN的话，暂且称为Spatial batch normalization，那么它的输入和输出的shape是（N,C,H,W）。我们的方法是讲这个4维数组reshape成（N'，D'）形式的2维数组，然后直接调用之前写好的batchnorm_forward方法来归一化，归一化之后再reshape回（N,C,H,W）。问题就是如何reshape了。我们现在希望在统计上，同一个通道的同一张图上不同位置和不同图像之间的特征是相对一致的。所以我们应该对一个通道中的数据做归一化。因此reshape后的N'=N*H*W，D'=C。由此可得代码：

```
    N, C, H, W = x.shape
    x2 = x.transpose(0, 2, 3, 1).reshape((N*H*W, C))
    out, cache = batchnorm_forward(x2, gamma, beta, bn_param)
    out = out.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 6.18949336e-16  5.99520433e-16 -1.22124533e-16]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

Spatial batch normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

注解

这里和前面是一样的了，先reshape然后调用batchnorm_backward计算，然后把结果reshape回来。代码如下：

```
    N, C, H, W = dout.shape
    dout2 = dout.transpose(0, 2, 3, 1).reshape(N*H*W, C)
    dx, dgamma, dbeta = batchnorm_backward(dout2, cache)
    dx = dx.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

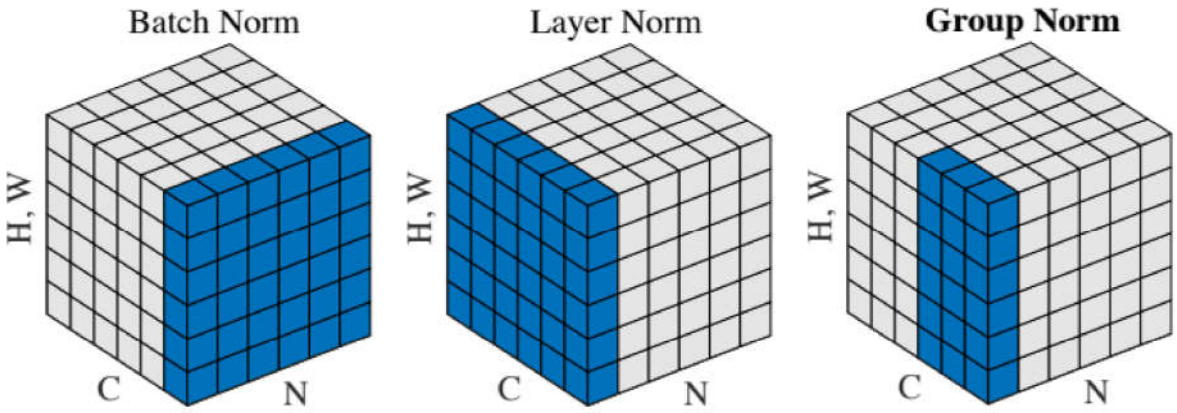
```
dx error:  2.786648201640115e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.



Visual comparison of the normalization techniques discussed so far (image edited from [5])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]-- after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* -- this truly is still an ongoing and excitingly active field of research!

[4][Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.] (https://arxiv.org/pdf/1607.06450.pdf)

[5][Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).] (https://arxiv.org/abs/1803.08494)

[6] [N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.](https://ieeexplore.ieee.org/abstract/document/1467360/)

Group normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

解释

Group normalization 就是将C分成C/G个组，每组内做Layer Normalization，所以我们模仿layernorm_forward的代码得到：

```
    N,C,H,W = x.shape
    x_group = np.reshape(x, (N, G, C//G, H, W)) #按G将C分组
    mean = np.mean(x_group, axis=(2,3,4), keepdims=True) #均值
    var = np.var(x_group, axis=(2,3,4), keepdims=True) #方差
    x_groupnorm = (x_group-mean)/np.sqrt(var+eps) #归一化
    x_norm = np.reshape(x_groupnorm, (N,C,H,W)) #还原维度
    out = x_norm*gamma+beta #还原C
    cache = (G, x, x_norm, mean, var, beta, gamma, eps)
```

```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x_g.mean(axis=1))
print('  Stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out_g.mean(axis=1))
print('  Stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [-2.14643118e-16  5.25505565e-16  2.65528340e-16 -3.38618023e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

Spatial group normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

解释

同样reshape之后模仿layernorm_backward得 代码如下：

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

    # 计算dx_group，(N, G, C // G, H, W)
    # dx_groupnorm
    dx_norm = dout * gamma
    dx_groupnorm = dx_norm.reshape((N, G, C // G, H, W))
    # dvar
    x_group = x.reshape((N, G, C // G, H, W))
    dvar = np.sum(dx_groupnorm * -1.0 / 2 * (x_group - mean) / (var + eps) ** (3.0 / 2), axis=
(2,3,4), keepdims=True)
    # dmean
    N_GROUP = C//G*H*W
    dmean1 = np.sum(dx_groupnorm * -1.0 / np.sqrt(var + eps), axis=(2,3,4), keepdims=True)
    dmean2_var = dvar * -2.0 / N_GROUP * np.sum(x_group - mean, axis=(2,3,4), keepdims=True)
    dmean = dmean1 + dmean2_var
    # dx_group
    dx_group1 = dx_groupnorm * 1.0 / np.sqrt(var + eps)
    dx_group2_mean = dmean * 1.0 / N_GROUP
    dx_group3_var = dvar * 2.0 / N_GROUP * (x_group - mean)
    dx_group = dx_group1 + dx_group2_mean + dx_group3_var

    # 还原C得到dx
    dx = dx_group.reshape((N, C, H, W))
```

```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
#You should expect errors of magnitudes between 1e-12~1e-07
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  7.413109622045623e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```