前面我们已经介绍了神经网络的一些常见的知识，这些神经网络其实都是全卷积神经网络，也就是层与层之间是全连接的，那么从今天开始我们就要开始一些新的内容，我们从卷积神经网络开始。

卷积神经网络和全连接的神经网络非常类似，它们都是由神经元组成，神经元中具有学习能力的权重和偏差，每个神经元都得到一些输入数据，进行内积运算后再进行激活函数运算。整个网络依旧是一个可导的评分函数：该函数的输入是原始的图像像素，输出是不同类别的评分。在最后一层（往往是全连接层），网络依旧是一个损失函数（比如SVM或Softmax），并且在神经网络中我们实现的各种技巧和要点依旧适用于卷积神经网络。

有一点不同的是CNN的各层中的神经元是3维排列的，宽度、高度和深度（这里的深度指的是激活数据体的第三个维度，而不是整个网络的深度，整个网络的深度指的是网络的层数）。

关于卷积神经网络（CNN）的介绍，前有很多大神对其进行了介绍，我相信他们的总结已经够我们了解CNN的基础知识了。

但是，为了我们能够更好的理解我们的代码，这里我们还是需要简要介绍下。

1 CNN基础层

CNN主要由三种类型的层构成：卷积层、池化层和全连接层。

全连接层在全连接神经网络中已经介绍，这里我们就不再赘述。

1.1 卷积层

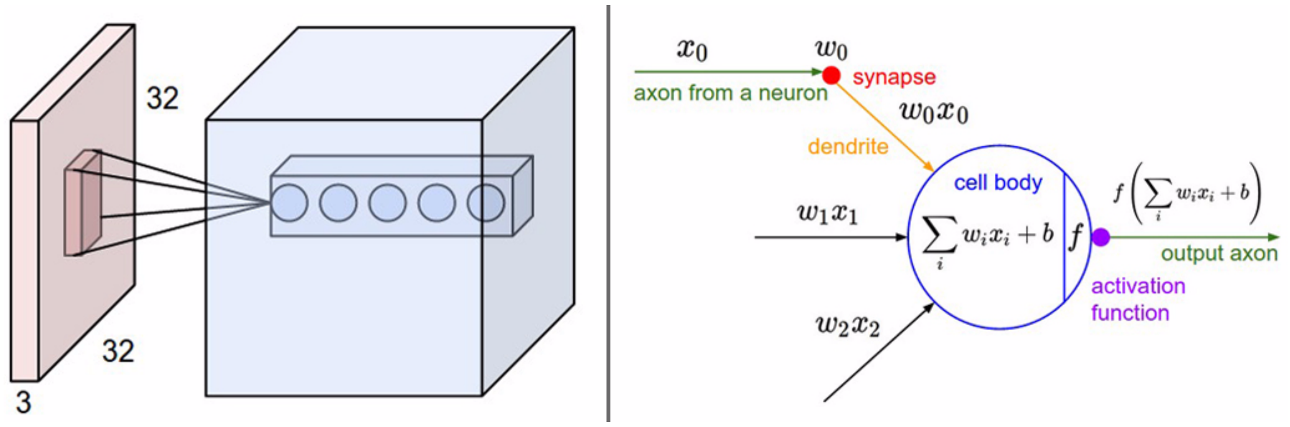
卷积层是构建卷积神经网络的核心层，它产生了网络中大部分的计算量。

1.1.1 前向传播

在前向传播的时候，让每个滤波器都在输入数据的宽度和高度上滑动（更精确地说是卷积），然后计算整个滤波器和输入数据任一处的内积。

卷积运算本质上就是滤波器和输入数据的局部区域做点积。卷积层的常用实现方式就是利用这一点，将卷积层的前向传播变成一个巨大的矩阵乘法。

卷积层中的神经元与输入层的一个**局部区域**相连，每个神经元都计算自己与输入层相连的小区域与自己权重的内积，卷积层会计算所有神经元的输出，如下图：



左图：红色是输入数据体，蓝色的部分是第一个卷积层中的神经元。卷积层中的每个神经元都只是与输入数据体的一个局部在空间上相连，但是与输入数据体的所有深度维度全部相连（所有颜色通道）。在深度方向上有多个神经元（本例中有5个），它们都接受输入数据的同一块区域（感受野相同）。
右图：神经元保持不变，它们还是计算权重和输入的内积，然后进行激活函数运算。

那么，在numpy中，这是如何实现的呢？

在真正实现之前，我们先看个小的例子：

- 一个位于(x,y)的深度列（或纤维）将会是**X[x,y,:]**。
- 在深度为d处的深度切片，或激活图应该是**X[:,:,d]**。

卷积层例子：假设输入数据体X的尺寸**X.shape:(11,11,4)**，不使用零填充（$P = 0$），滤波器的尺寸是$F = 5$，步长$S = 2$。那么输出数据体的空间尺寸就是(11-5)/2+1=4，即输出数据体的宽度和高度都是4。那么在输出数据体中的激活映射（称其为**V**）看起来就是下面这样（在这个例子中，只有部分元素被计算）：

- **V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0**
- **V[1,0,0] = np.sum(X[2:7,:5,:] * W0) + b0**
- **V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0**
- **V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0**

这是对一个滤波器在x方向上的操作，注意这里**一个滤波器的参数是共享的（都是W0 b0）**，其中W0的形状是(5,5,4)，b0的形状是(1,)

下面我们根据上述过程来实现一个完整的前向传播。
代码如下：

```
def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.
```

```python
    The input consists of N data points, each with C channels, height H and width
    W. We convolve each input with F different filters, where each filter spans
    all C channels and has height HH and width HH.
    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
      - 'stride': The number of pixels between adjacent receptive fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be used to zero-pad the input.
    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None
    #############################################################################
    # TODO: Implement the convolutional forward pass.                           #
    # Hint: you can use the function np.pad for padding.                        #
    #############################################################################
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) / stride
    W_out = 1 + (W + 2 * pad - WW) / stride
    out = np.zeros((N, F, H_out, W_out))

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', constant_values=0)
    for i in range(H_out):
        for j in range(W_out):
            x_pad_masked = x_pad[:, :, i * stride:i * stride + HH, j * stride:j * stride + WW]
            for k in range(F):
                out[:, k, i, j] = np.sum(x_pad_masked * w[k, :, :, :], axis=(1, 2, 3))
                # out[:, : , i, j] = np.sum(x_pad_masked * w[:, :, :, :], axis=(1,2,3))

                # for k in range(F):
                # out[:, k, :, :] = out[:, k, :, :] + b[k]
    out = out + (b)[None, :, None, None]
    # pass
    ##############################################################################
    #                             END OF YOUR CODE                               #
    ##############################################################################
    cache = (x, w, b, conv_param)
    return out, cache
```

代码中需要计算卷积层的输出，关于输出的解释如下：

卷积神经网络中，3个超参数控制着输出数据体的尺寸：

1. 深度。它和使用的滤波器的数量一致，而每个滤波器在输入数据中寻找一些**不同**的东西；
2. 步长。步长是滑动滤波器时移动的像素数，如步长为1时，滤波器每次移动1个像素；
3. 零填充。用0在输入数据边缘处进行填充，这在保持输入数据体在空间上的尺寸时很有效。

那么有了这三个超参数后，得到输出数据的尺寸是多少呢？
输出数据体的空间尺寸为：(W-F+2P)/S+1。其中W是输入数据尺寸，F是感受野尺寸（卷积核大小），S是步长，P是零填充数量。

1.1.2 前向传播检验

同样，我们来检验下。

```python
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 1e-8
print ('Testing conv_forward_naive')
print ('difference: ', rel_error(out, correct_out))
```

输出结果如下：

```
Testing conv_forward_naive
difference:  2.21214764175e-08
```

误差还是很小的，当然，我们这里只是实现了基本的前向传播。

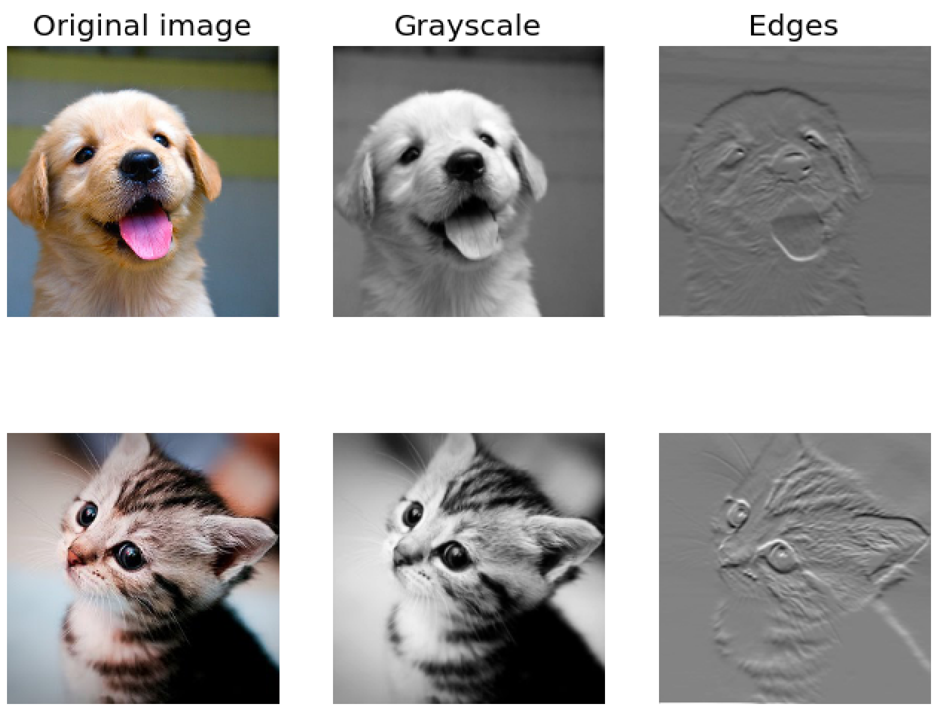
另外一个检验我们的前向传播过程是直接将卷积层作用于图像，人工设置滤波器，然后可视化卷积层的输出。

这里两个滤波器分别实现了灰度转换和边缘检测。
代码如下：

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

img_size = 200   # Make this smaller if it runs too slow
```

```python
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'),cmap='gray')
    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()
```

得到的结果如下：



1.1.3 反向传播

卷积操作的反向传播（同时对于数据和权重）还是一个卷积（但是是和空间上反转的滤波器）。

反向传播的时候，都要计算每个神经元对它的权重的梯度，但是需要把同一个滤波器的所有神经元对权重的梯度累加，这样就得到了对共享权重的梯度。这样，每个滤波器只更新一个权重。

关于CNN反向传播，这里有一个蛮好的解释：CNN反向传播，我们直接截取了最后的结论部分。
其中：

$$z^l = a^{l-1} * W^l + b$$

dx的计算：

$$\nabla a^{l-1}=\frac{\partial J(W,b)}{\partial a^{l-1}}=\frac{\partial J(W,b)}{\partial z^l}\frac{\partial z^l}{\partial a^{l-1}}=\delta^l\frac{\partial z^l}{\partial a^{l-1}}$$

如：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{11} & \delta_{12} & 0 \\ 0 & \delta_{21} & \delta_{22} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{pmatrix} = \begin{pmatrix} \nabla a_{11} & \nabla a_{12} & \nabla a_{13} \\ \nabla a_{21} & \nabla a_{22} & \nabla a_{23} \\ \nabla a_{31} & \nabla a_{32} & \nabla a_{33} \end{pmatrix}$$

dw的计算：

$$\frac{\partial J(W,b)}{\partial W^l}=\frac{\partial J(W,b)}{\partial z^l}\frac{\partial z^l}{\partial W^l}=\delta^l * rot180(a^{l-1})$$

db的计算：

$$\frac{\partial J(W,b)}{\partial b^l}=\sum_{u,v}(\delta^l)_{u,v}$$

实现代码如下：

```python
def conv_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a convolutional layer.
    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None
    #############################################################################
    # TODO: Implement the convolutional backward pass.                          #
    #############################################################################
    x, w, b, conv_param = cache

    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) / stride
    W_out = 1 + (W + 2 * pad - WW) / stride

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', constant_values=0)
    dx = np.zeros_like(x)
    dx_pad = np.zeros_like(x_pad)
    dw = np.zeros_like(w)
    db = np.zeros_like(b)

    db = np.sum(dout, axis=(0, 2, 3))

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', constant_values=0)
    for i in range(H_out):
        for j in range(W_out):
            x_pad_masked = x_pad[:, :, i * stride:i * stride + HH, j * stride:j * stride + WW]
            for k in range(F):  # compute dw
                dw[k, :, :, :] += np.sum(x_pad_masked * (dout[:, k, i, j])[:, None, None, None], axis=0)
            for n in range(N):  # compute dx_pad
                dx_pad[n, :, i * stride:i * stride + HH, j * stride:j * stride + WW] += np.sum((w[:, :, :, :] *
                                                       (dout[n, :, i, j])[:,
                                                       None, None, None]),
                                                      axis=0)
    dx = dx_pad[:, :, pad:-pad, pad:-pad]
    # pass

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

1.1.4 反向传播误差检验

检测代码如下：

```python
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 1e-9'
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))
```

得到结果如下：

```
Testing conv_backward_naive function
dx error:  2.63064435662e-09
dw error:  5.71755216084e-10
db error:  2.67587467777e-11
```

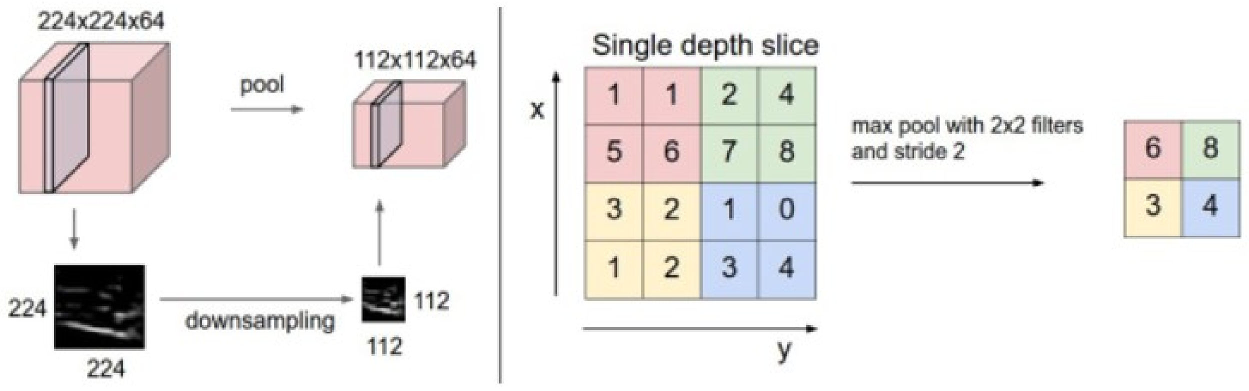
满足要求。

1.2 最大池化层

池化层有最大池化层、平均池化层等，这里我们只介绍最大池化层。

1.2.1 前向传播

最大池化层的前向传播过程如下：



汇聚层在输入数据体的每个深度切片上，独立地对其进行空间上的降采样。左边：本例中，输入数据体尺寸[224x224x64]被降采样到了[112x112x64]，采取的滤波器尺寸是2，步长为2，而深度不变。右边：最常用的降采样操作是取最大值，也就是最大汇聚，这里步长为2，每个取最大值操作是从4个数字中选取（即2x2的方块区域中）。

实现代码如下：

```python
def max_pool_forward_naive(x, pool_param):
  """
  A naive implementation of the forward pass for a max pooling layer.
  Inputs:
  - x: Input data, of shape (N, C, H, W)
  - pool_param: dictionary with the following keys:
    - 'pool_height': The height of each pooling region
    - 'pool_width': The width of each pooling region
    - 'stride': The distance between adjacent pooling regions
  Returns a tuple of:
  - out: Output data
  - cache: (x, pool_param)
  """
  out = None
  #############################################################################
  # TODO: Implement the max pooling forward pass                              #
  #############################################################################
  N, C, H, W = x.shape
  HH, WW, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
  H_out = int((H-HH)/stride+1)
  W_out = int((W-WW)/stride+1)
  out = np.zeros((N,C,H_out,W_out))
  for i in range(H_out):
        for j in range(W_out):
            x_masked = x[:,:,i*stride : i*stride+HH, j*stride : j*stride+WW]
            out[:,:,i,j] = np.max(x_masked, axis=(2,3))
  #pass
  #############################################################################
  #                             END OF YOUR CODE                              #
  #############################################################################
  cache = (x, pool_param)
  return out, cache
```

1.2.2 前向传播检验

实现代码如下：

```
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 around 1e-8.
print ('Testing max_pool_forward_naive function:')
print ('difference: ', rel_error(out, correct_out))
```

结果如下：

```
Testing max_pool_forward_naive function:
```

```
difference:  4.16666651573e-08
```

1.2.3 反向传播

反向传播dx的所有子矩阵的各个池化局域的值放在之前做前向传播算法得到的最大值的位置，如下：

$$\delta_k^l = \begin{pmatrix} 2 & 8 \\ 4 & 6 \end{pmatrix} \text{--------------->} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \end{pmatrix}$$

$$\delta_k^{l-1} = \frac{\partial J(W,b)}{\partial a_k^{l-1}} \frac{\partial a_k^{l-1}}{\partial z_k^{l-1}} = upsample(\delta_k^l) \odot \sigma'(z_k^{l-1})$$

实现代码如下：

```python
def max_pool_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a max pooling layer.
    Inputs:
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.
    Returns:
    - dx: Gradient with respect to x
    """
    dx = None
    #############################################################################
    # TODO: Implement the max pooling backward pass                             #
    #############################################################################
    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_out = (H - HH) / stride + 1
    W_out = (W - WW) / stride + 1
    dx = np.zeros_like(x)

    for i in xrange(H_out):
        for j in xrange(W_out):
            x_masked = x[:, :, i * stride: i * stride + HH, j * stride: j * stride + WW]
            max_x_masked = np.max(x_masked, axis=(2, 3))
            temp_binary_mask = (x_masked == (max_x_masked)[:, :, None, None])
            dx[:, :, i * stride: i * stride + HH, j * stride: j * stride + WW] += temp_binary_mask * (dout[:, :, i, j])[
                                                                                      :, :, None, None]
    # pass
    #############################################################################
    #                             END OF YOUR CODE                              #
    #############################################################################
    return dx

```

1.2.4 反向传播梯度检验

检验代码如下：

```python
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 around 1e-12
print ('Testing max_pool_backward_naive function:')
print ('dx error: ', rel_error(dx, dx_num))
```

结果如下：

```
Testing max_pool_backward_naive function:
dx error:  3.27563441384e-12
```

当然，上述过程只是帮助我们理解了CNN的整个传输过程，在接下来的具体的网络上，我们将采用计算速度更快的层（准确率是一样的）。这里我们就不对这些代码进行列举了，具体的代码可参阅前文提到的github地址。

不将它列举出来还有一个原因，是新手不需要掌握，只需要根据上述代码知道原理即可，因为现在的框架那么发达，我们学会使用框架岂不美哉～～

为防止混淆，效率更高的层我们分别定义为 `conv_forward_fast` `conv_backward-fast` `max_pool_forward_fast` `max_pool_backward_fast`

2 复合层

这是我自己的命名，在cs231n中将其称为“三明治”层。

在实际中，卷积层一般是和激活函数层、池化层一起使用，因此这里我们可以直接将它们连接成一个个的复杂的层。

2.1 卷积+激活函数

2.1.1 层的实现

代码如下：

```
def conv_relu_forward(x, w, b, conv_param):
    """
    A convenience layer that performs a convolution followed by a ReLU.
    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer

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

```python


def conv_relu_backward(dout, cache):
    """
    Backward pass for the conv-relu convenience layer.
    """
    conv_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db
```

2.1.2 误差检验

代码如下：

```python
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)

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))

```

运行结果如下：

```
Testing conv_relu:
dx error: 8.63479795379e-09
dw error: 1.11675179506e-09
db error: 1.76423143581e-11
```

2.2 卷积+激活+池化

2.2.1 层的实现

代码如下：

```python
def conv_relu_pool_forward(x, w, b, conv_param, pool_param):
    """
    Convenience layer that performs a convolution, a ReLU, and a pool.
    Inputs:
    - x: Input to the convolutional layer
    - w, b, conv_param: Weights and parameters for the convolutional layer
    - pool_param: Parameters for the pooling layer
    Returns a tuple of:
    - out: Output from the pooling layer
    - cache: Object to give to the backward pass
    """
    a, conv_cache = conv_forward_fast(x, w, b, conv_param)
    s, relu_cache = relu_forward(a)
    out, pool_cache = max_pool_forward_fast(s, pool_param)
    cache = (conv_cache, relu_cache, pool_cache)
    return out, cache


def conv_relu_pool_backward(dout, cache):

    conv_cache, relu_cache, pool_cache = cache
    ds = max_pool_backward_fast(dout, pool_cache)
    da = relu_backward(ds, relu_cache)
    dx, dw, db = conv_backward_fast(da, conv_cache)
    return dx, dw, db
```

2.2.2 误差检验

代码如下：

```python
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)
```

```python
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)

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))

```

得到的输出结果如下：

```
Testing conv_relu_pool
dx error: 9.39697106034e-08
dw error: 6.33831387553e-08
db error: 2.33627428737e-10
```

3 卷积神经网络模型构建

我们来构建一个三层的卷积神经网络模型，网络结构如下：
conv--> relu--> 2*2 max pool--> affine--> relu--> affine--> softmax

代码如下：

```python
class ThreeLayerConvNet(object):
    def __init__(self,input_dim=(3,32,32),num_filters=32,filter_size=7,
                 hidden_dim=100,num_classes=10,weight_scale=1e-3,reg=0.0,
                 dtype=np.float32):
        self.params={}
        self.reg=reg
        self.dtype=dtype

        #参数初始化
        C,H,W=input_dim
        self.params['W1']=weight_scale*np.random.randn(num_filters,C,filter_size,filter_size)
        self.params['b1']=np.zeros(num_filters)
        self.params['W2']=weight_scale*np.random.randn((H/2)*(W/2)*num_filters,hidden_dim) #与卷积层的输出尺寸有关
        self.params['b2']=np.zeros(hidden_dim)
        self.params['W3']=weight_scale*np.random.randn(hidden_dim,num_classes)
        self.params['b3']=np.zeros(num_classes)

        for k, v in self.params.iteritems():
            self.params[k] = v.astype(dtype)

    def loss(self,X,y=None):
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        W3, b3 = self.params['W3'], self.params['b3']

        filter_size = W1.shape[2]
        conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}

        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

        scores = None

        #前向传播
        conv_forward_out_1, cache_forward_1 = conv_relu_pool_forward(X, self.params['W1'], self.params['b1'],
                                                                      conv_param, pool_param)
        affine_forward_out_2, cache_forward_2 = affine_forward(conv_forward_out_1, self.params['W2'], self.params['b2'])
        affine_relu_2, cache_relu_2 = relu_forward(affine_forward_out_2)
        scores, cache_forward_3 = affine_forward(affine_relu_2, self.params['W3'], self.params['b3'])

        if y is None:
            return scores

        loss,grads=0,{}

        #反向传播
        loss, dout = softmax_loss(scores, y)

        #增加正则项
        loss += self.reg * 0.5 * (
        np.sum(self.params['W1'] ** 2) + np.sum(self.params['W2'] ** 2) + np.sum(self.params['W3'] ** 2))

        dX3, grads['W3'], grads['b3'] = affine_backward(dout, cache_forward_3)
        dX2 = relu_backward(dX3, cache_relu_2)
        dX2, grads['W2'], grads['b2'] = affine_backward(dX2, cache_forward_2)
        dX1, grads['W1'], grads['b1'] = conv_relu_pool_backward(dX2, cache_forward_1)

        grads['W3'] = grads['W3'] + self.reg * self.params['W3']
        grads['W2'] = grads['W2'] + self.reg * self.params['W2']
        grads['W1'] = grads['W1'] + self.reg * self.params['W1']

        return loss, grads

```

3.1 损失函数检验

当我们使用softmax作为损失函数时，随机权重得到的损失函数值（没有正则项）为ln(C)，C表示种类个数。

当增加正则项时，损失函数值将增加。

我们来检验下看看，代码如下：

```python

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)

```

得到的结果如下：

```
Initial loss (no regularization): 2.3025846903
Initial loss (with regularization): 2.50874836489
```

满足我们的要求。

3.2 梯度检验

代码如下：

```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
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)
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])))
```

得到的结果如下：

```
W1 max relative error: 4.383184e-04
W2 max relative error: 4.104758e-03
W3 max relative error: 6.594642e-05
b1 max relative error: 1.075895e-05
b2 max relative error: 2.542041e-07
b3 max relative error: 1.513297e-09
```

相对误差相比我们之前的几次作业稍有些大（以前大都是1e-8），但也是可以理解的，因为随着网络层数的加深，误差在一点点的积累，也会越来越大。

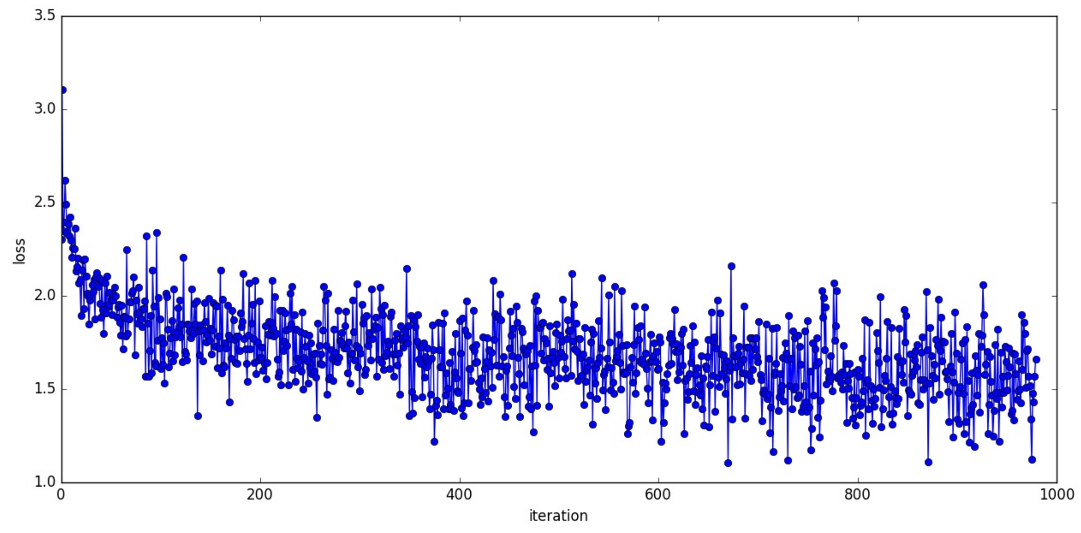
总之，现在我们已经完成了前面的铺垫工作，下面就可以用我们的cifar10数据进行相关操作了。

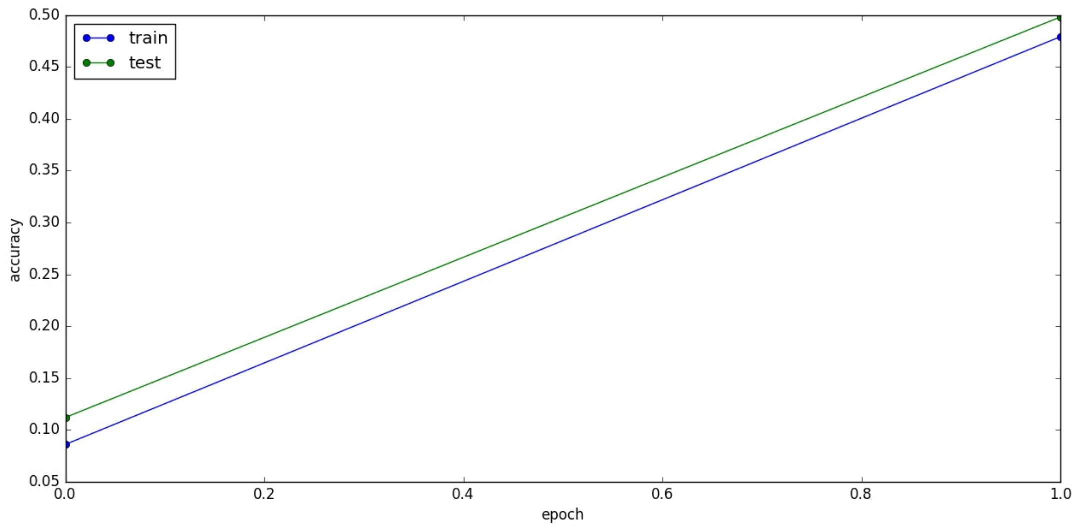
4 训练

为了节省时间，我们只训练了1个epoch。
代码如下：

```python
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()
plt.figure(1)
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',label='train')
plt.plot(solver.val_acc_history, '-o',label='test')
plt.legend(loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')

plt.gcf().set_size_inches(15, 15,forward=True)
plt.savefig('train.jpg')
plt.show()

```

得到结果如下：





损失函数在逐渐减小。

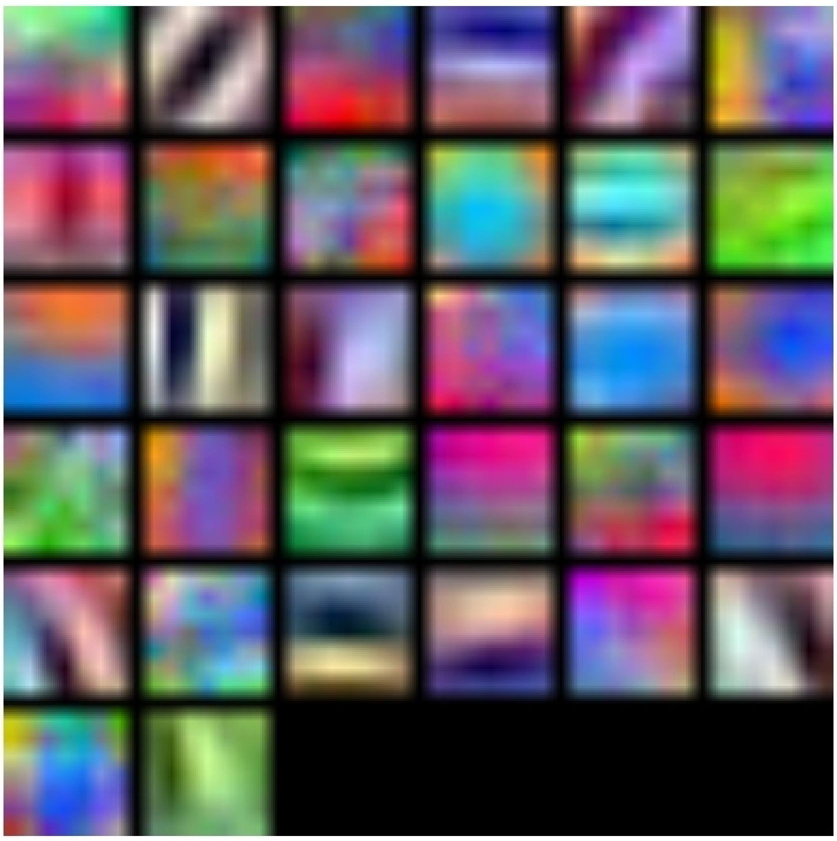
当然，为了提高预测准确率，还可以增加epoch，继续训练。

下面我们还可视化下我们的权重，看下我们的卷积神经网络到底学到了些什么？

可视化代码如下：

```
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,forward=True)
plt.savefig('weight.jpg')
plt.show()
```

得到的结果如下：



还是比较模糊的，这里我们设置的滤波器个数有点少，为了学到更多的特征，读者可以多增加几个滤波器。

5 空间BN层

在上一次的作业中，我们实现了BN层，并将其运用到了全连接网络中，取得了不错的效果。这节我们就来实现下用于卷积神经网络的BN层。

由于平面BN输入和输出是二维的，卷积之后得到的四维的，所以这里我们需要进行一个小的转化，即(N,C,H,W)转化为(N×H×W,C)。

4.1 前向传播

代码如下：

```
def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """
    Computes the forward pass for spatial batch normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance. momentum=0 means
that
        old information is discarded completely at every time step, while
        momentum=1 means that new information is never incorporated. The
        default of momentum=0.9 should work well in most situations.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of
features

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None


 ########################################################################
#####
    # TODO: Implement the forward pass for spatial batch normalization.
      #
    #
      #
    # HINT: You can implement spatial batch normalization using the
vanilla     #
    # version of batch normalization defined above. Your implementation
should   #
    # be very short; ours is less than five lines.
       #

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

 ########################################################################
#####
```

```
    #                             END OF YOUR CODE                             #
    ###########################################################################

    return out, cache
```

4.2 反向传播

代码如下：

```
def spatial_batchnorm_backward(dout, cache):
    """
    Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    ###########################################################################
    # TODO: Implement the backward pass for spatial batch normalization.       #
    #                                                                         #
    # HINT: You can implement spatial batch normalization using the vanilla    #
    # version of batch normalization defined above. Your implementation should  #
    # be very short; ours is less than five lines.                            #
    ###########################################################################
    N, C, H, W = dout.shape
    dx_temp, dgamma, dbeta = batchnorm_backward_alt(dout.transpose(0, 3, 2, 1).reshape((N * H * W, C)), cache)
    dx = dx_temp.reshape(N, W, H, C).transpose(0, 3, 2, 1)
    # pass
    #############################################################################
    #                             END OF YOUR CODE                              #
    #############################################################################

    return dx, dgamma, dbeta

```

5 总结

当然，关于卷积神经网络肯定是还没有结束，还可以调整超参数来逐步提高在验证集上面的准确率，最终可以达到90+%（当然，可能要花很多时间，而且还需要做数据增强）。

关于神经网络的训练，这里有魏秀参博士的一篇博客，可参阅。