FullyConnectedNet

2019年4月9日 12:08

好的终于开始CNN了！
上来第一件事，报错！

In [1]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
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))))
```

```
run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel
```

然后回到anaconda pormpt输入python setup.py build_ext结果又报错
Module not found Cython（图没截到，应该是这个错）
先pip install Cython，结束后再次输入python setup.py build_ext --inplace
又报了一个错……找不到vcvarsall.bat……
上网查了贼多资料……最后总结一句话，安装VS2019，总之安装最新的应该就没有错……

折腾半天终于build好了，打开jupyter notebook又出问题了，我的Chrome界面一直空白显示不出东西……
然后打开anaconda的安装目录\envs\cs231n\Lib\site-packages\notebook\notebookapp.py
ctrl+F定位到def init_mime_overrides(self):
然后把最后一行注释掉改成
mimetypes.add_type('application/javascript','.js')
再次运行jupyter notebook要是页面还是空白ctrl+F5刷新即可！

In [2]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
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))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

抱头痛哭!

In [3]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(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,))
```

加载数据集

下面先完成cs231n/layers.py的affine_forward

```
def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.

    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """
    out = None
    ###########################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You    #
    # will need to reshape the input into rows.
    ###########################################################################
    out = None
    reshaped_x = np.reshape(x,(x.shape[0],-1))
    out = reshaped_x.dot(w) + b
    # out = wx + b
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b)
    # cache把该函数的输入值存为元组，方便在之后的反向传播中用到。
    return out, cache
```

输入：

x：N个样本，k个维度

w：D等于x中所有维度的乘积

b：偏移量

输出：output=w*x+b

为了后向传播方便计算梯度，把该层的x，w，b等信息保存到cache里面

```
In [26]: # Test the affine_forward function

         num_inputs = 2
         input_shape = (4, 5, 6)
         output_dim = 3

         input_size = num_inputs * np.prod(input_shape)
         #np.prod没指定坐标轴的情况下，计算所有元素乘积
         weight_size = output_dim * np.prod(input_shape)

         x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
         # np.linspace返回一个均匀分布,*input_shape返回（4,5,6）
         w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
         b = np.linspace(-0.3, 0.1, num=output_dim)

         out, _ = affine_forward(x, w, b)
         correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                                 [ 3.25553199,  3.5141327,   3.77273342]])

         # Compare your output with ours. The error should be around 1e-9.
         print('Testing affine_forward function:')
         print('difference: ', rel_error(out, correct_out))

         Testing affine_forward function:
         difference:  9.769847728806635e-10
```

验证affine_forward的结果

完成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)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., 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.
    ###########################################################################
    N,D = x.shape[0],w.shape[0]
    dx = dout.dot(w.T).reshape(x.shape)
    dw = x.reshape((N,D)).T.dot(dout)
    db = np.sum(dout, axis=0)
    # 对b求导梯度不变
    # 每一列求和
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
    return dx, dw, db
```

输入：

dout：根据链式法则反向传播计算出来的当前层之前的导数

cache：在forward中存下来的当前层的参数

输出：

dx：相当于dout里面包含wx+b，dout对x求导等于dout*w（求导后还原成之前的shape）

dw：同理等于dout*x

db：同理

In [28]:
```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
# lambda x相当于定义一个函数f(x)该函数eval……的第一个参数是一个函数，返回值为affine
# _forward的第一个参数即output，看eval_numerical……发现该函数输入的第一个参数即为
# 函数f
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around 1e-10
print('Testing affine_backward function:')
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 affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

验证backward的结果

下面完成relu_forward

```
def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.
    ###########################################################################
    out = np.maximum(0,x)
    ###########################################################################
    #                             END OF YOUR CODE
    ###########################################################################
    cache = x
    return out, cache
```

显然

In [29]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be around 5e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

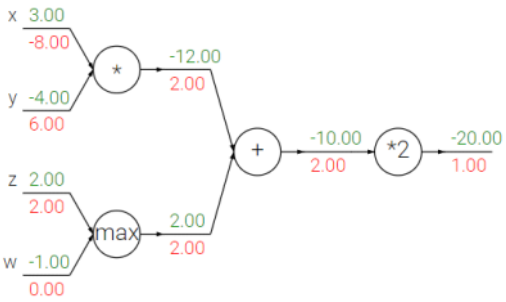
验证结果

完成relu_backward

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

    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.
    #############################################################################
    dx = dout * (x>0)
    # max gate要么为0要么不变
    #############################################################################
    #                             END OF YOUR CODE
    #############################################################################
    return dx
```

x>0是一个shape和x一样，元素为0和1的矩阵，x值大于0则为1，小于等于0则为0

根据链式法则的反向传播，在一个max gate里，求导要么为0，要么不变。



An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.

In [30]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)
# 具有标准正态分布
dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 3e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

验证结果

下面完成cs231n/layers_until的affine_relu_forward和backward

cs231n里面把这个叫做sandwich layers比较形象，就是把这些不同类型的层按顺序连接在一起

拿一个出来当例子看

```
def affine_relu_forward(x, w, b):
    """
    Convenience layer that perorms 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
```

先是一个affine层，输入x，w，b返回a，fc_cache，a再作为参数被relu层调用返回out和relu_cache，最后将两个cache都存入一个cachelist

里，返回out和cache
最终就是输入x，w，b返回经过affine层和relu层得到的output和cache

In [31]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

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

print('Testing affine_relu_forward:')
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 affine_relu_forward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

验证结果

In [32]:
```
np.random.seed(231)
# 生成seed231，每次从231里面取随机数都是一样的
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

然后完成cs231n/classifiers/fc_net.py的towlayernet

```
def __init__(self, input_dim=3*32*32, hidden_dim=100, num_classes=10,
             weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.

    Inputs:
    - input_dim: An integer giving the size of the input
    - hidden_dim: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
       initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
    """
    self.params = {}
    self.reg = reg
```

```
        ############################################################################
        # TODO: Initialize the weights and biases of the two-layer net. Weights    #
        # should be initialized from a Gaussian with standard deviation equal to   #
        # weight_scale, and biases should be initialized to zero. All weights and  #
        # biases should be stored in the dictionary self.params, with first layer  #
        # weights and biases using the keys 'W1' and 'b1' and second layer weights #
        # and biases using the keys 'W2' and 'b2'.                                 #
        ############################################################################
        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,))
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################
```

先完成初始化部分

```
    def loss(self, X, y=None):
        """
        Compute loss and gradient for a minibatch of data.

        Inputs:
        - X: Array of input data of shape (N, d_1, ..., d_k)
        - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

        Returns:
        If y is None, then run a test-time forward pass of the model and return:
        - scores: Array of shape (N, C) giving classification scores, where
          scores[i, c] is the classification score for X[i] and class c.

        If y is not None, then run a training-time forward and backward pass and
        return a tuple of:
        - loss: Scalar value giving the loss
        - grads: Dictionary with the same keys as self.params, mapping parameter
          names to gradients of the loss with respect to those parameters.
        """
        scores = None

        ############################################################################
        # TODO: Implement the forward pass for the two-layer net, computing the    #
        # class scores for X and storing them in the scores variable.              #
        ############################################################################
        W1 = self.params['W1']
        b1 = self.params['b1']
        W2 = self.params['W2']
        b2 = self.params['b2']
        A1,A1_cache = affine_forward(X,W1,b1)
        H1,H1_cache = relu_forward(A1)
        A2,A2_cache = affine_forward(H1,W2,b2)
        scores = A2
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        # If y is None then we are in test mode so just return scores
        if y is None:
            return scores

        loss, grads = 0, {}

        ############################################################################
        # TODO: Implement the backward pass for the two-layer net. Store the loss  #
        # in the loss variable and gradients in the grads dictionary. Compute data #
        # loss using softmax, and make sure that grads[k] holds the gradients for  #
        # self.params[k]. Don't forget to add L2 regularization!                   #
        #                                                                          #
        # NOTE: To ensure that your implementation matches ours and you pass the   #
        # automated tests, make sure that your L2 regularization includes a factor #
        # of 0.5 to simplify the expression for the gradient.                      #
        ############################################################################
        loss,dscores = softmax_loss(scores,y)
        loss += 0.5 * self.reg * (np.sum(W1*W1) + (np.sum(W2*W2)))
        dH1,dW2,db2 = affine_backward(dscores,A2_cache)
        grads['W2'] = dW2 + self.reg * W2
        grads['b2'] = db2;
        dA1 = relu_backward(dH1,H1_cache)
        dX,dW1,db1 = affine_backward(dA1,A1_cache)
        grads['W1'] = dW1 + self.reg * W1
        grads['b1'] = db1
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return loss, grads
```

第一个部分的代码类似上面讲的sandwich只不过这里是affine-relu-afine

接下来，这里用到的loss是softmax，在之前的assignment1里面讲过，

这里记住求出来的关于W的梯度要加上正则化部分的梯度就ok

In [33]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)
# 随机生成一个 (N,) array，值为不超过C的整数

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'
```

```
print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 8.18e-07
W2 relative error: 2.85e-08
b1 relative error: 1.09e-09
b2 relative error: 7.76e-10
```

这一步还是验证模型的准确性

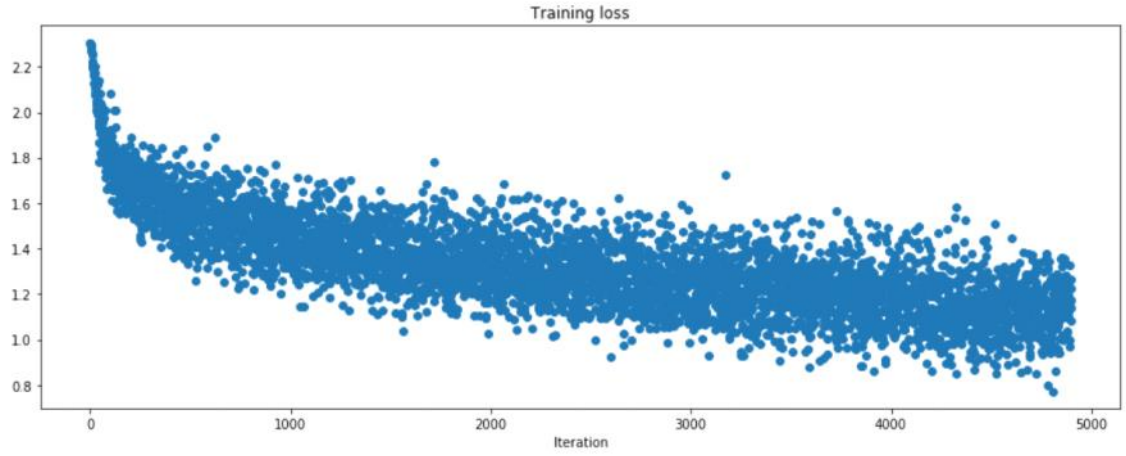
In [34]:
```
model = TwoLayerNet()
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                          #
################################################################################
data  = get_CIFAR10_data()
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()
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

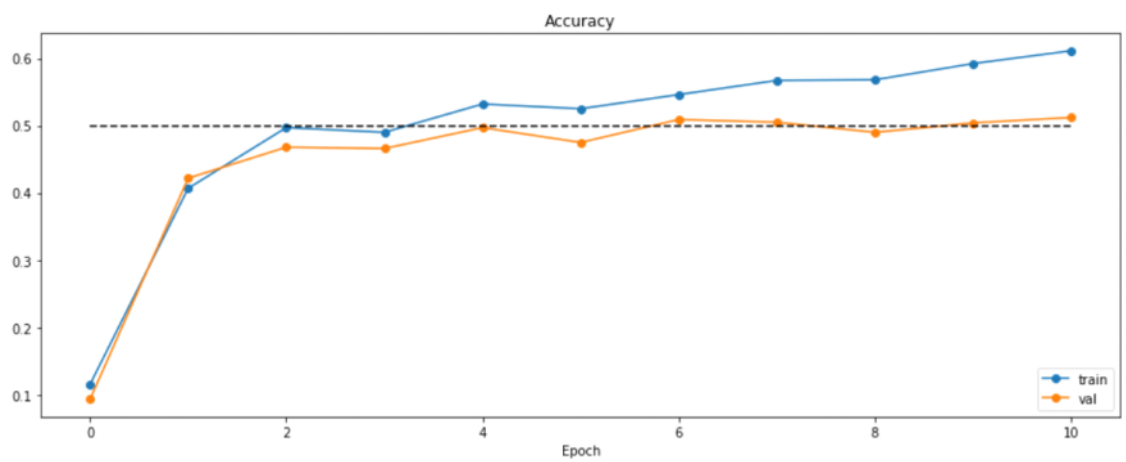
solver用法在solver.py里面有解释

```
(Epoch 6 / 10) train acc: 0.546000; val_acc: 0.509000
(Iteration 3001 / 4900) loss: 1.304489
(Iteration 3101 / 4900) loss: 1.346667
(Iteration 3201 / 4900) loss: 1.325510
(Iteration 3301 / 4900) loss: 1.392728
(Iteration 3401 / 4900) loss: 1.402001
(Epoch 7 / 10) train acc: 0.567000; val_acc: 0.505000
(Iteration 3501 / 4900) loss: 1.319024
(Iteration 3601 / 4900) loss: 1.153287
(Iteration 3701 / 4900) loss: 1.180922
(Iteration 3801 / 4900) loss: 1.093164
(Iteration 3901 / 4900) loss: 1.135902
(Epoch 8 / 10) train acc: 0.568000; val_acc: 0.490000
(Iteration 4001 / 4900) loss: 1.191735
(Iteration 4101 / 4900) loss: 1.359396
(Iteration 4201 / 4900) loss: 1.227283
(Iteration 4301 / 4900) loss: 1.024113
(Iteration 4401 / 4900) loss: 1.327583
(Epoch 9 / 10) train acc: 0.592000; val_acc: 0.504000
(Iteration 4501 / 4900) loss: 0.963330
(Iteration 4601 / 4900) loss: 1.445619
(Iteration 4701 / 4900) loss: 1.007542
(Iteration 4801 / 4900) loss: 1.005175
(Epoch 10 / 10) train acc: 0.611000; val_acc: 0.512000
```

截一部分结果

In [35]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





接下来完成fc_net.py里面的fullyconnectednet

```
class FullyConnectedNet(object):
    """
    A fully-connected neural network with an arbitrary number of hidden layers,
    ReLU nonlinearities, and a softmax loss function. This will also implement
    dropout and batch normalization as options. For a network with L layers,
    the architecture will be

    {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

    where batch normalization and dropout are optional, and the {...} block is
    repeated L - 1 times.

    Similar to the TwoLayerNet above, learnable parameters are stored in the
    self.params dictionary and will be learned using the Solver class.
    """

    def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
                 dropout=0, use_batchnorm=False, reg=0.0,
                 weight_scale=1e-2, dtype=np.float32, seed=None):
        """
        Initialize a new FullyConnectedNet.

        def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
                     dropout=0, use_batchnorm=False, reg=0.0,
                     weight_scale=1e-2, dtype=np.float32, seed=None):
            """
            Initialize a new FullyConnectedNet.

            Inputs:
            - hidden_dims: A list of integers giving the size of each hidden layer.
            - input_dim: An integer giving the size of the input.
            - num_classes: An integer giving the number of classes to classify.
            - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
               the network should not use dropout at all.
            - use_batchnorm: Whether or not the network should use batch normalization.
            - reg: Scalar giving L2 regularization strength.
            - weight_scale: Scalar giving the standard deviation for random
               initialization of the weights.
            - dtype: A numpy datatype object; all computations will be performed using
               this datatype. float32 is faster but less accurate, so you should use
               float64 for numeric gradient checking.
            - seed: If not None, then pass this random seed to the dropout layers. This
               will make the dropout layers deteriminstic so we can gradient check the
               model.
            """
            self.use_batchnorm = use_batchnorm
            self.use_dropout = dropout > 0
            self.reg = reg
            self.num_layers = 1 + len(hidden_dims)
            self.dtype = dtype
            self.params = {}


            ############################################################################
            # TODO: Initialize the parameters of the network, storing all values in    #
            # the self.params dictionary. Store weights and biases for the first layer #
            # in W1 and b1; for the second layer use W2 and b2, etc. Weights should be #
            # initialized from a normal distribution with standard deviation equal to  #
            # weight_scale and biases should be initialized to zero.                   #
            #                                                                          #
            # When using batch normalization, store scale and shift parameters for the #
            # first layer in gamma1 and beta1; for the second layer use gamma2 and     #
            # beta2, etc. Scale parameters should be initialized to one and shift      #
            # parameters should be initialized to zero.                                #
            ############################################################################
            all_dims = [input_dim, *hidden_dims, num_classes]
            for i in range(len(all_dims)):
                if(i == 0):
                    continue
                self.params['W'+str(i)] = np.random.normal(0,weight_scale,(all_dims[i-1],all_dims[i]))
                self.params['b'+str(i)] = np.zeros((all_dims[i],))
                if use_batchnorm:
                    self.params['gamma'+str(i)] = np.ones((1,all_dims[i]))
                    self.params['beta'+str(i)] = np.zeros((1,all_dims[i]))
            ############################################################################
            #                             END OF YOUR CODE                             #
            ############################################################################
```

```
        # When using dropout we need to pass a dropout_param dictionary to each
        # dropout layer so that the layer knows the dropout probability and the mode
        # (train / test). You can pass the same dropout_param to each dropout layer.
        self.dropout_param = {}
        if self.use_dropout:
            self.dropout_param = {'mode': 'train', 'p': dropout}
            if seed is not None:
                self.dropout_param['seed'] = seed

        # With batch normalization we need to keep track of running means and
        # variances, so we need to pass a special bn_param object to each batch
        # normalization layer. You should pass self.bn_params[0] to the forward pass
        # of the first batch normalization layer, self.bn_params[1] to the forward
        # pass of the second batch normalization layer, etc.
        self.bn_params = []
        if self.use_batchnorm:
            self.bn_params = [{'mode': 'train'} for i in range(self.num_layers - 1)]

        # Cast all parameters to the correct datatype
        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)
```

```
    def loss(self, X, y=None):
        """
        Compute loss and gradient for the fully-connected net.

        Input / output: Same as TwoLayerNet above.
        """
        X = X.astype(self.dtype)
        mode = 'test' if y is None else 'train'

        # Set train/test mode for batchnorm params and dropout param since they
        # behave differently during training and testing.
        if self.use_dropout:
            self.dropout_param['mode'] = mode
        if self.use_batchnorm:
            for bn_param in self.bn_params:
                bn_param['mode'] = mode

        scores = None
        ############################################################################
        # TODO: Implement the forward pass for the fully-connected net, computing  #
        # the class scores for X and storing them in the scores variable.          #
        #                                                                          #
        # When using dropout, you'll need to pass self.dropout_param to each       #
        # dropout forward pass.                                                    #
        #                                                                          #
        # When using batch normalization, you'll need to pass self.bn_params[0] to #
        # the forward pass for the first batch normalization layer, pass           #
        # self.bn_params[1] to the forward pass for the second batch normalization #
        # layer, etc.                                                              #
        ############################################################################
        fc_mix_cache = {}
        if self.use_dropout:
            dp_cache = {}
        out = X
        for i in range(self.num_layers - 1):
            w,b = self.params["W%d"%(i+1,)],self.params["b%d"%(i+1,)]
            if self.use_batchnorm:
                gamma = self.params["gamma%d"%(i+1,)]
#                 print(gamma)
                beta = self.params["beta%d"%(i+1,)]
                out,fc_mix_cache[i] = affine_bn_relu_forward(out,w,b,gamma
                 ,beta,self.bn_params[i])
            else:
                out,fc_mix_cache[i] = affine_relu_forward(out,w,b)
            if self.use_dropout:
                out,dp_cache[i] = dropout_forward(out,self.dropout_param)
        w = self.params["W%d" % (self.num_layers,)]
        b = self.params["b%d" % (self.num_layers,)]
#         print(self.params)
        out ,out_cache = affine_forward(out,w,b)
#         out,fc_mix_cache[i+1] = affine_forward(out,w,b)
        scores = out
```

```
        # If test mode return early
        if mode == 'test':
            return scores

        loss, grads = 0.0, {}
        ############################################################################
        # TODO: Implement the backward pass for the fully-connected net. Store the #
        # loss in the loss variable and gradients in the grads dictionary. Compute #
        # data loss using softmax, and make sure that grads[k] holds the gradients #
        # for self.params[k]. Don't forget to add L2 regularization!               #
        #                                                                          #
        # When using batch normalization, you don't need to regularize the scale   #
        # and shift parameters.                                                    #
        #                                                                          #
        # NOTE: To ensure that your implementation matches ours and you pass the   #
        # automated tests, make sure that your L2 regularization includes a factor #
        # of 0.5 to simplify the expression for the gradient.                      #
        ############################################################################

        loss,dout = softmax_loss(scores,y)
        loss += 0.5 * self.reg * np.sum(self.params["W%d"%(self.num_layers,)]**2)
        dout,dw,db = affine_backward(dout,out_cache)
        grads["W%d"%(self.num_layers,)]=dw+self.reg*self.params["W%d"%(self.num_layers,)]
        grads["b%d"%(self.num_layers,)] =db
        for i in range(self.num_layers - 1):
            ri = self.num_layers - 2 - i
            loss += 0.5*self.reg*np.sum(self.params["W%d"%(ri+1,)]**2)
            if self.use_dropout:
                dout = dropout_backward(dout,dp_cache[ri])
            if self.use_batchnorm:
                dout,dw,db,dgamma,dbeta = affine_bn_relu_backward(dout,fc_mix_cache[ri])
                grads["gamma%d"%(ri+1,)] = dgamma
#                 print((grads["gamma%d"%(ri+1,)]).shape)
#                 print((self.params["gamma%d"%(ri+1,)]).shape)
#                 print(dgamma)
                grads["beta%d"%(ri+1,)] = dbeta
            else:
                dout,dw,db = affine_relu_backward(dout,fc_mix_cache[ri])
            grads["W%d"%(ri+1,)] = dw+self.reg*self.params["W%d"%(ri+1,)]
            grads["b%d"%(ri+1,)] = db
#         print(grads)
        if self.use_batchnorm:
            grads["gamma%d"%(self.num_layers,)]=self.params["gamma%d"%(self.num_layers,)]
            grads["beta%d"%(self.num_layers,)]=self.params["beta%d"%(self.num_layers,)]
        ############################################################################
        #                                                 END OF YOUR CODE         #
        ############################################################################

        return loss, grads
```

这里就question1,2,3一起讲了

仔细看代码会发现这里初始化和求loss以及优化都有三种方法，一种是原始方法，一种使用了batchnorm，一种使用了dropout

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

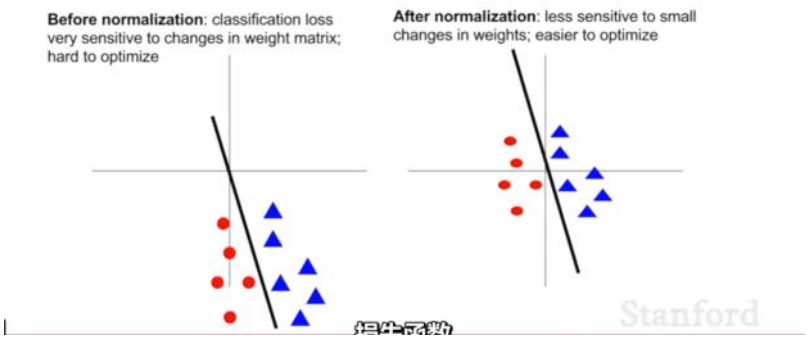
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

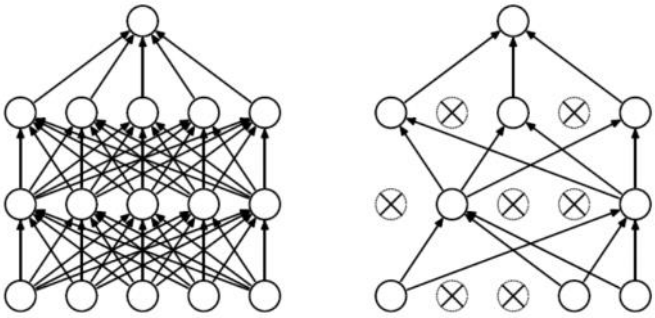
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

这是batchnorm的正向传播的计算公式

先让数据接近高斯分布再加上两个可学习的参数去学习该层的特征分布



左图是没有进行规范化处理的，右图是经过规范化处理之后的数据，
大多数情况下左图收敛速度会小于右图，且准确度没有右图好。



(a) Standard Neural Net (b) After applying dropout.

dropout，随机失活，网络越复杂越容易过拟合，使用dropout让隐藏层的部分神经元失效，可以在一定程度上防止过拟合的发生。
前面那段代码主要就是初始化各个层的w和b和gamma和beta然后在前向传播的时候调用他们，算出output和cache，然后再后向传播算出每一层各个参数的梯度。

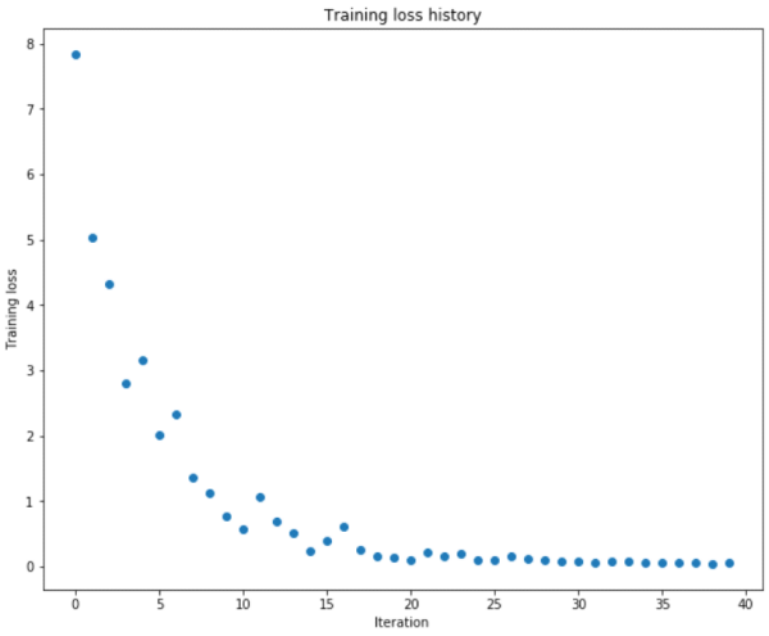
In [52]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

检验各层W和b算出来的值

In [54]:
```
# TODO: Use a three-layer Net to overfit 50 training examples.

num_train = 50
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'],
}

weight_scale = 3e-2
learning_rate = 1e-3
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 7.840284
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.079000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.094000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.132000
(Epoch 3 / 20) train acc: 0.600000; val_acc: 0.125000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.124000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.134000
(Iteration 11 / 40) loss: 0.564432
(Epoch 6 / 20) train acc: 0.820000; val_acc: 0.125000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.138000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.137000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.146000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.149000
(Iteration 21 / 40) loss: 0.104197
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.144000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.138000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.141000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.138000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.144000
(Iteration 31 / 40) loss: 0.069173
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.143000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.136000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.140000
```



拿50个样本来训练和检验

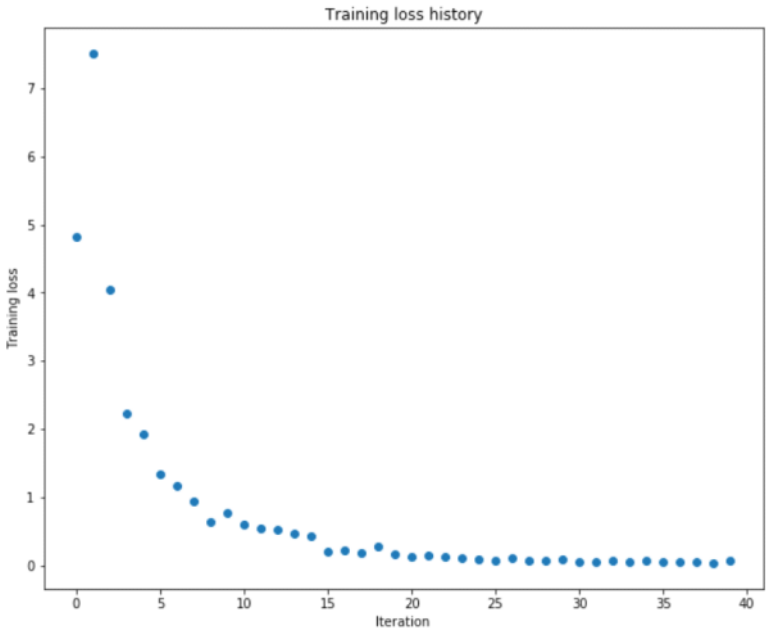
model = FullyConnectedNet([100, 100]　　看这里

hidden_dims只有两个元素这是一个三层网络

下面是一个5层的网络

In [55]:
```
# TODO: Use a five-layer Net to overfit 50 training examples.

num_train = 50
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'],
}

learning_rate = 8e-3
weight_scale = 5e-2
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 4.812411
(Epoch 0 / 20) train acc: 0.140000; val_acc: 0.088000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.133000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.130000
(Epoch 3 / 20) train acc: 0.720000; val_acc: 0.117000
(Epoch 4 / 20) train acc: 0.880000; val_acc: 0.136000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.153000
(Iteration 11 / 40) loss: 0.605515
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.132000
(Epoch 7 / 20) train acc: 0.940000; val_acc: 0.130000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.147000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.148000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.156000
(Iteration 21 / 40) loss: 0.117019
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.156000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.151000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.152000
(Iteration 31 / 40) loss: 0.048046
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.150000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.149000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.149000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.151000
```



可以看到增加层数可以提高准确率

之前的优化策略都是“sgd”，详见fullyconnected里面的update rule，下面试几个新的优化策略。
这里用了sgd+momentum Adam RMSprop


opt2


opt1

放两个图不知道会不会动，但是在我自己的电脑上双击能显示，再放一个链接解释吧
http://cs231n.github.io/neural-networks-3/

In [57]:
```
from cs231n.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

老规矩，先验证准确性

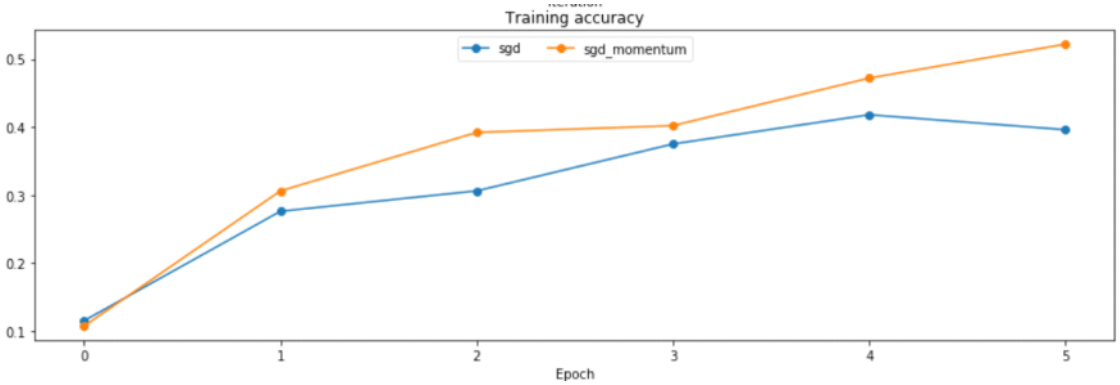
In [71]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
```

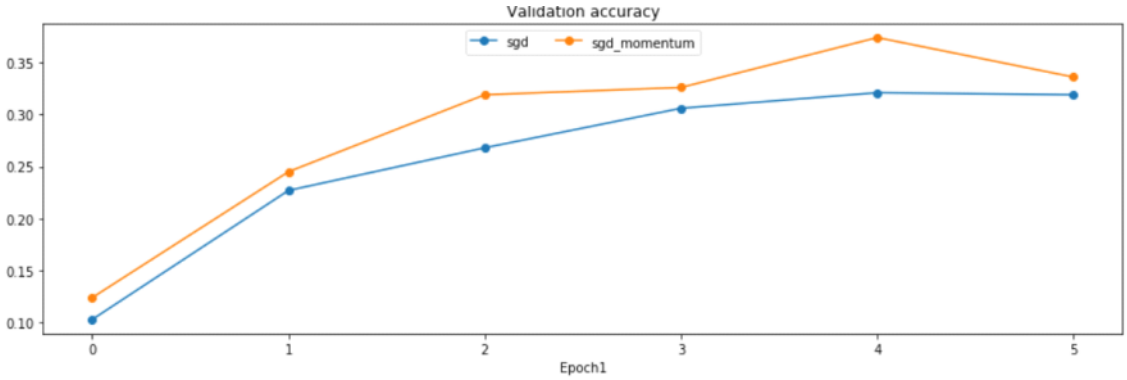
```
plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch1')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
```

```
plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch1')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.598676
(Epoch 0 / 5) train acc: 0.115000; val_acc: 0.103000
(Iteration 11 / 200) loss: 2.172998
(Iteration 21 / 200) loss: 2.171712
(Iteration 31 / 200) loss: 1.959542
(Epoch 1 / 5) train acc: 0.276000; val_acc: 0.227000
(Iteration 41 / 200) loss: 2.076739
(Iteration 51 / 200) loss: 1.989500
(Iteration 61 / 200) loss: 2.006675
(Iteration 71 / 200) loss: 1.825188
(Epoch 2 / 5) train acc: 0.306000; val_acc: 0.268000
(Iteration 81 / 200) loss: 1.891575
(Iteration 91 / 200) loss: 1.878704
(Iteration 101 / 200) loss: 1.869473
(Iteration 111 / 200) loss: 1.807945
(Epoch 3 / 5) train acc: 0.375000; val_acc: 0.306000
(Iteration 121 / 200) loss: 1.659942
(Iteration 131 / 200) loss: 1.809584
(Iteration 141 / 200) loss: 1.692280
(Iteration 151 / 200) loss: 1.822620
(Epoch 4 / 5) train acc: 0.418000; val_acc: 0.321000
(Iteration 161 / 200) loss: 1.645426
(Iteration 171 / 200) loss: 1.612308
(Iteration 181 / 200) loss: 1.737271
(Iteration 191 / 200) loss: 1.794228
(Epoch 5 / 5) train acc: 0.396000; val_acc: 0.319000
```

```
running with  sgd_momentum
(Iteration 1 / 200) loss: 2.401017
(Epoch 0 / 5) train acc: 0.107000; val_acc: 0.124000
(Iteration 11 / 200) loss: 2.095937
(Iteration 21 / 200) loss: 2.160691
(Iteration 31 / 200) loss: 1.964303
(Epoch 1 / 5) train acc: 0.306000; val_acc: 0.245000
(Iteration 41 / 200) loss: 2.028297
(Iteration 51 / 200) loss: 1.891669
(Iteration 61 / 200) loss: 1.825353
(Iteration 71 / 200) loss: 1.754680
(Epoch 2 / 5) train acc: 0.392000; val_acc: 0.319000
(Iteration 81 / 200) loss: 1.555130
(Iteration 91 / 200) loss: 1.575670
(Iteration 101 / 200) loss: 1.811315
(Iteration 111 / 200) loss: 1.737880
(Epoch 3 / 5) train acc: 0.402000; val_acc: 0.326000
(Iteration 121 / 200) loss: 1.723961
(Iteration 131 / 200) loss: 1.700424
(Iteration 141 / 200) loss: 1.324919
(Iteration 151 / 200) loss: 1.522778
(Epoch 4 / 5) train acc: 0.472000; val_acc: 0.374000
(Iteration 161 / 200) loss: 1.431371
(Iteration 171 / 200) loss: 1.368986
(Iteration 181 / 200) loss: 1.545932
(Iteration 191 / 200) loss: 1.351163
(Epoch 5 / 5) train acc: 0.522000; val_acc: 0.336000
```





4000个样本5层网络

Sgd_momentum效果可以看出来优于sgd

In [65]:
```
# Test RMSProp implementation; you should see errors less than 1e-7
from cs231n.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

In [69]:
```
# Test Adam implementation; you should see errors around 1e-7 or less
from cs231n.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])
```

```
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

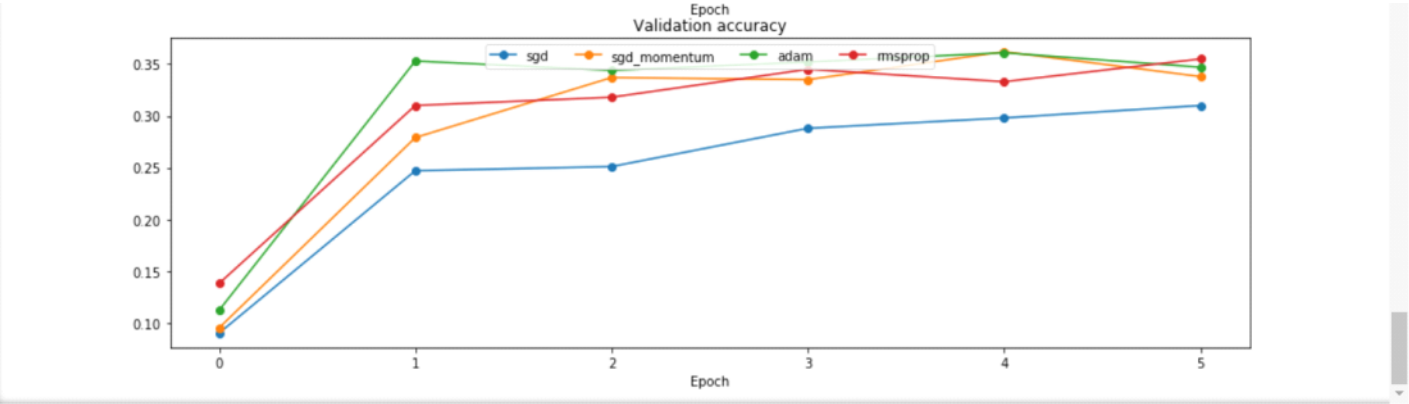
```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

In [70]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')
```

```
for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
(Epoch 2 / 5) train acc: 0.409000; val_acc: 0.318000
(Iteration 81 / 200) loss: 1.694086
(Iteration 91 / 200) loss: 1.621452
(Iteration 101 / 200) loss: 1.416712
(Iteration 111 / 200) loss: 1.573677
(Epoch 3 / 5) train acc: 0.465000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.580463
(Iteration 131 / 200) loss: 1.788524
(Iteration 141 / 200) loss: 1.589166
(Iteration 151 / 200) loss: 1.570254
(Epoch 4 / 5) train acc: 0.474000; val_acc: 0.333000
(Iteration 161 / 200) loss: 1.510541
(Iteration 171 / 200) loss: 1.460289
(Iteration 181 / 200) loss: 1.268078
(Iteration 191 / 200) loss: 1.501438
(Epoch 5 / 5) train acc: 0.540000; val_acc: 0.355000
```

这里截一部分



套路和上面一样，就不解释了

最下面那个不想做了，才50%左右，感觉没意思，看看后面有没有什么大招吧。

Batchnorm

2019年4月21日 14:04

Batchnorm其实就是对每一个隐藏层做一个批量归一化，来提高收敛速度和效果。

这里直接从第三个cell开始了。

先完成cs231n/layers.py的batchnorm_forward

```
if mode == 'train':
    #######################################################################
    # TODO: Implement the training-time forward pass for batch norm.      #
    # Use minibatch statistics to compute the mean and variance, use      #
    # these statistics to normalize the incoming data, and scale and      #
    # shift the normalized data using gamma and beta.                     #
    #                                                                     #
    # You should store the output in the variable out. Any intermediates  #
    # that you need for the backward pass should be stored in the cache   #
    # variable.                                                           #
    #                                                                     #
    # You should also use your computed sample mean and variance together #
    # with the momentum variable to update the running mean and running   #
    # variance, storing your result in the running_mean and running_var   #
    # variables.                                                          #
    #######################################################################
    sample_mean = np.mean(x,axis = 0)
    sample_var = np.var(x,axis = 0)
    x_hat = (x - sample_mean) / (np.sqrt(sample_var + eps))
    out = gamma * x_hat + beta
    cache = (x,sample_mean, sample_var, x_hat, eps, gamma, beta)
    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1- momentum) * sample_var
    #######################################################################
    #                                END OF YOUR CODE                     #
    #######################################################################
```

这里就截取了自己写的部分

```
elif mode == 'test':
    #######################################################################
    # TODO: Implement the test-time forward pass for batch normalization. #
    # Use the running mean and variance to normalize the incoming data,   #
    # then scale and shift the normalized data using gamma and beta.      #
    # Store the result in the out variable.                               #
    #######################################################################
    out = (x - running_mean) * gamma / (np.sqrt(running_var + eps)) + beta
    #######################################################################
    #                                END OF YOUR CODE                     #
    #######################################################################
```

In [5]:

```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before batch normalization:')
print('  means: ', a.mean(axis=0))
print('  stds: ', a.std(axis=0))

# Means should be close to zero and stds close to one
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = batchnorm_forward(a, np.ones(D3), np.zeros(D3), {'mode': 'train'})
print('  mean: ', a_norm.mean(axis=0))
print('  std: ', a_norm.std(axis=0))

# Now means should be close to beta and stds close to gamma
gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print('After batch normalization (nontrivial gamma, beta)')
print('  means: ', a_norm.mean(axis=0))
print('  stds: ', a_norm.std(axis=0))
```

```
Before batch normalization:
  means:  [ -2.3814598  -13.18038246   1.91780462]
  stds:  [27.18502186 34.21455511 37.68611762]
After batch normalization (gamma=1, beta=0)
  mean:  [5.32907052e-17 7.04991621e-17 4.22578639e-17]
  std:  [0.99999999 1.         1.        ]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [0.99999999 1.99999999 2.99999999]
```

In [6]:
```
# 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.
np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)
for t in range(50):
  X = np.random.randn(N, D1)
  a = np.maximum(0, X.dot(W1)).dot(W2)
  batchnorm_forward(a, gamma, beta, bn_param)
bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, 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 batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=0))
print('  stds: ', a_norm.std(axis=0))
```

```
After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:  [1.01531428 1.01238373 0.97819988]
```

完成batchnorm_backward

```
###############################################################################
# TODO: Implement the backward pass for batch normalization. Store the      #
# results in the dx, dgamma, and dbeta variables.                             #
###############################################################################
x, mean, var, x_hat, eps, gamma, beta = cache
dbeta = np.sum(dout, axis=0, keepdims=True)
dgamma = np.sum(dout * x_hat, axis=0, keepdims=True)
dx_hat = dout*gamma
dx_hat_numerator = dx_hat / np.sqrt(var + eps)
dx_hat_denominator = np.sum(dx_hat * (x - mean), axis=0)
dx_1 = dx_hat_numerator
dvar = -0.5 * ((var + eps)**-1.5) * dx_hat_denominator
dmean = -1.0 * np.sum(dx_hat_numerator, axis=0) + \
        -2.0 * dvar * np.sum(x - mean, axis=0)
N = x.shape[0]
dx_var = dvar * 2.0 / N * (x - mean)
dx_mean = dmean * 1.0 / N
dx = dx_1 + dx_var + dx_mean
###############################################################################
#                                 END OF YOUR CODE                            #
###############################################################################
```

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

这是batchnorm反向传播求梯度的公式

上面的计算基本上是按步求下来的。

In [10]:
```
# Gradient check batchnorm backward pass
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

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

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

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = 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:  1.7029221415332113e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

验证梯度

下面那个batchnorm_backward_alt没有做.....

In [34]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64,
                            use_batchnorm=True)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 2.85e-06
W3 relative error: 3.92e-10
b1 relative error: 4.44e-08
b2 relative error: 2.00e-07
b3 relative error: 4.78e-11
beta1 relative error: 7.33e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 7.57e-09
gamma2 relative error: 1.96e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.11e-08
b1 relative error: 1.11e-08
b2 relative error: 2.36e-08
b3 relative error: 2.23e-10
beta1 relative error: 6.65e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 5.94e-09
gamma2 relative error: 4.14e-09
```

In [64]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
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'],
}

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=True)
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=False)

bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
bn_solver.train()
```

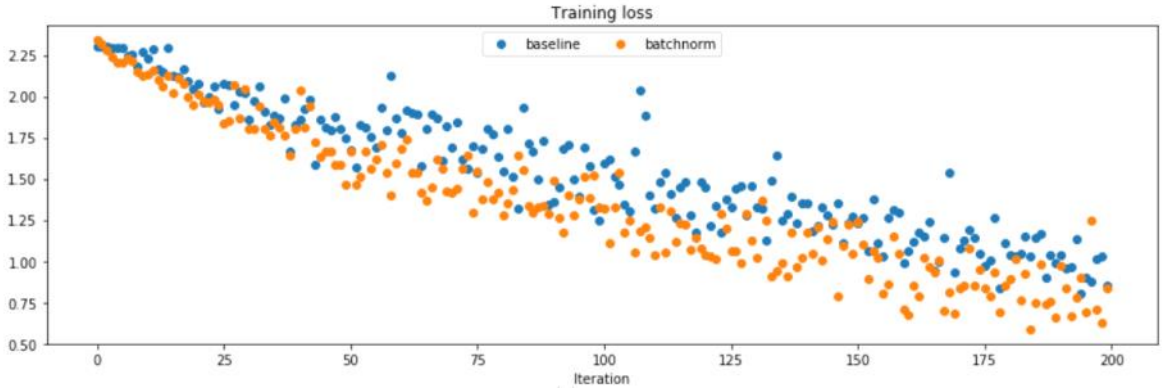
```
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
solver.train()
```

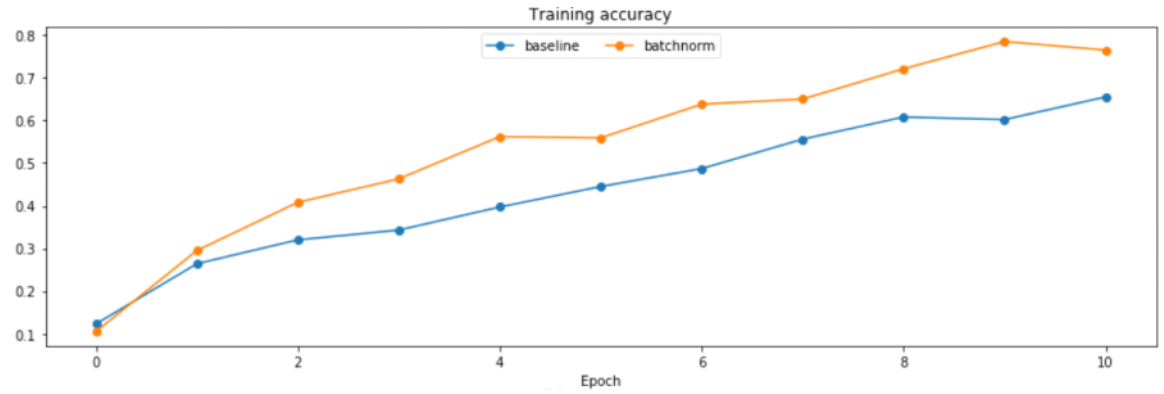
```
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.111000
(Epoch 1 / 10) train acc: 0.295000; val_acc: 0.236000
(Epoch 2 / 10) train acc: 0.408000; val_acc: 0.298000
(Epoch 3 / 10) train acc: 0.463000; val_acc: 0.299000
(Epoch 4 / 10) train acc: 0.562000; val_acc: 0.342000
(Epoch 5 / 10) train acc: 0.559000; val_acc: 0.321000
(Epoch 6 / 10) train acc: 0.638000; val_acc: 0.347000
(Epoch 7 / 10) train acc: 0.650000; val_acc: 0.309000
(Epoch 8 / 10) train acc: 0.721000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.785000; val_acc: 0.340000
(Epoch 10 / 10) train acc: 0.765000; val_acc: 0.313000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.123000; val_acc: 0.130000
(Epoch 1 / 10) train acc: 0.264000; val_acc: 0.212000
(Epoch 2 / 10) train acc: 0.320000; val_acc: 0.298000
(Epoch 3 / 10) train acc: 0.343000; val_acc: 0.275000
(Epoch 4 / 10) train acc: 0.397000; val_acc: 0.318000
(Epoch 5 / 10) train acc: 0.445000; val_acc: 0.314000
(Epoch 6 / 10) train acc: 0.487000; val_acc: 0.339000
(Epoch 7 / 10) train acc: 0.556000; val_acc: 0.305000
(Epoch 8 / 10) train acc: 0.608000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.602000; val_acc: 0.335000
(Epoch 10 / 10) train acc: 0.655000; val_acc: 0.298000
```

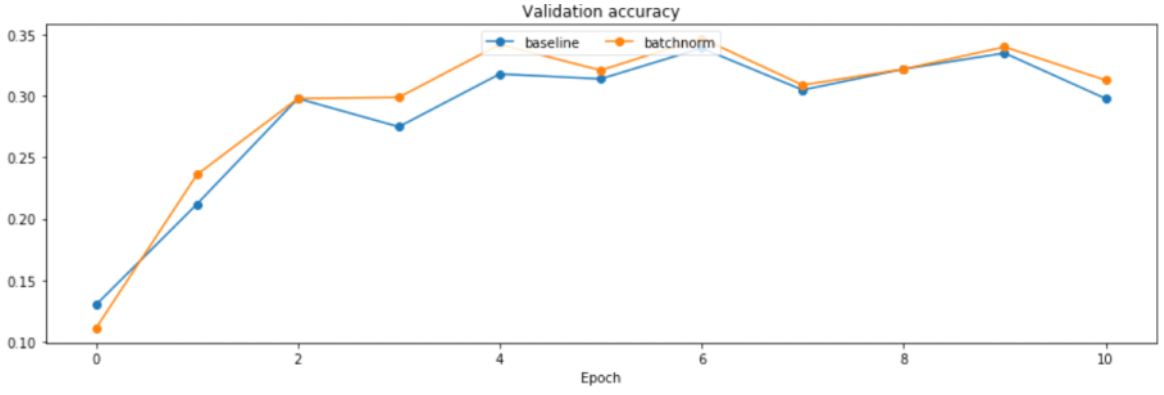
6层网络，1000个样本使用batchnorm和Adam

In [65]:
```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(solver.loss_history, 'o', label='baseline')
plt.plot(bn_solver.loss_history, 'o', label='batchnorm')

plt.subplot(3, 1, 2)
plt.plot(solver.train_acc_history, '-o', label='baseline')
plt.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

plt.subplot(3, 1, 3)
plt.plot(solver.val_acc_history, '-o', label='baseline')
plt.plot(bn_solver.val_acc_history, '-o', label='batchnorm')
```

```
for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```







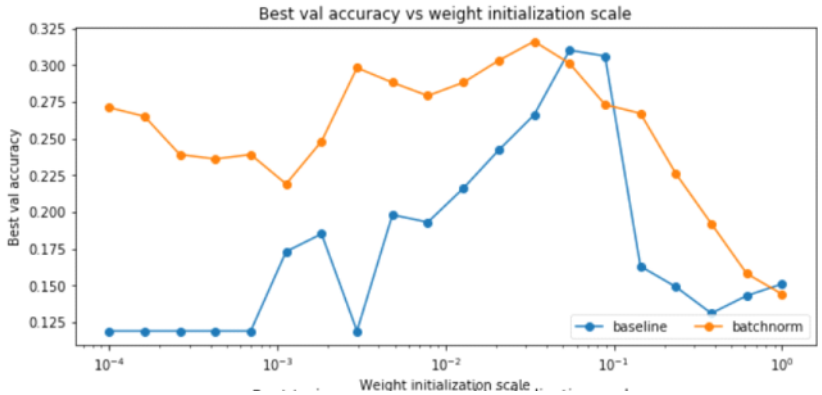
下面学习weight scale

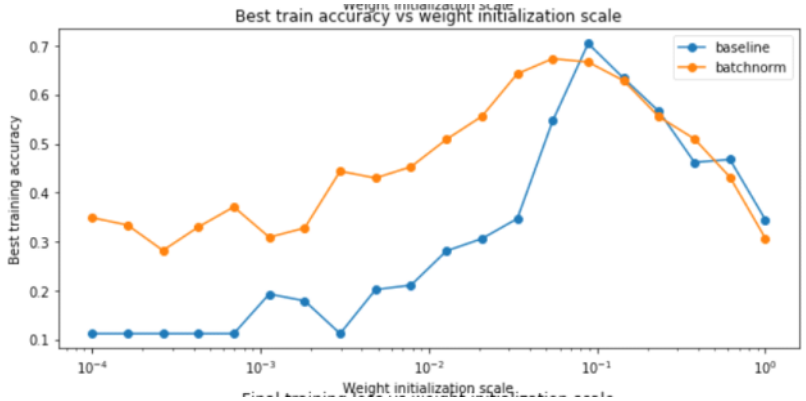
In [64]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
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'],
}

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=True)
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, use_batchnorm=False)

bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
bn_solver.train()
```

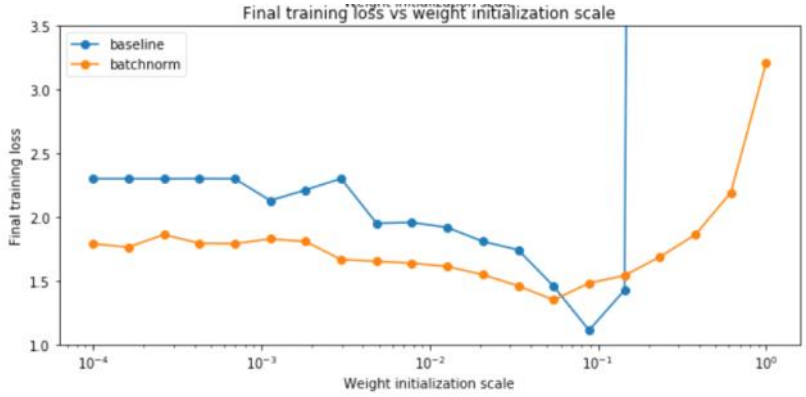
```
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=200)
solver.train()
```

```
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.105000; val_acc: 0.111000
(Epoch 1 / 10) train acc: 0.295000; val_acc: 0.236000
(Epoch 2 / 10) train acc: 0.408000; val_acc: 0.298000
(Epoch 3 / 10) train acc: 0.463000; val_acc: 0.299000
(Epoch 4 / 10) train acc: 0.562000; val_acc: 0.342000
(Epoch 5 / 10) train acc: 0.559000; val_acc: 0.321000
(Epoch 6 / 10) train acc: 0.638000; val_acc: 0.347000
(Epoch 7 / 10) train acc: 0.650000; val_acc: 0.309000
(Epoch 8 / 10) train acc: 0.721000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.785000; val_acc: 0.340000
(Epoch 10 / 10) train acc: 0.765000; val_acc: 0.313000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.123000; val_acc: 0.130000
(Epoch 1 / 10) train acc: 0.264000; val_acc: 0.212000
(Epoch 2 / 10) train acc: 0.320000; val_acc: 0.298000
(Epoch 3 / 10) train acc: 0.343000; val_acc: 0.275000
(Epoch 4 / 10) train acc: 0.397000; val_acc: 0.318000
(Epoch 5 / 10) train acc: 0.445000; val_acc: 0.314000
(Epoch 6 / 10) train acc: 0.487000; val_acc: 0.339000
(Epoch 7 / 10) train acc: 0.556000; val_acc: 0.305000
(Epoch 8 / 10) train acc: 0.608000; val_acc: 0.322000
(Epoch 9 / 10) train acc: 0.602000; val_acc: 0.335000
(Epoch 10 / 10) train acc: 0.655000; val_acc: 0.298000
```

In [65]:
```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(solver.loss_history, 'o', label='baseline')
plt.plot(bn_solver.loss_history, 'o', label='batchnorm')

plt.subplot(3, 1, 2)
plt.plot(solver.train_acc_history, '-o', label='baseline')
plt.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

plt.subplot(3, 1, 3)
plt.plot(solver.val_acc_history, '-o', label='baseline')
plt.plot(bn_solver.val_acc_history, '-o', label='batchnorm')

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```







每次画图的warning可以忽略，大概是因为xlabel内容是一样的造成的。

Dropout

2019年4月21日 15:21

这个assignment做完第一个，后面这两个就快的很了，基本上到后一半都是直接shift+enter飞过。所以batchnorm和这里的dropout就讲的少了些，主要看代码都很好理解。

完成dropout_forward

```
    if mode == 'train':
        ###########################################################################
        # TODO: Implement training phase forward pass for inverted dropout.     #
        # Store the dropout mask in the mask variable.                                                  #
        ###########################################################################
        keep_prob = 1 - p
        mask = (np.random.rand(*x.shape) < keep_prob) / keep_prob
        out = mask * x
        ###########################################################################
        #                                      END OF YOUR CODE                   #
        ###########################################################################
    elif mode == 'test':
        ###########################################################################
        # TODO: Implement the test phase forward pass for inverted dropout.     #
        ###########################################################################
        out = x
        ###########################################################################
        #                                      END OF YOUR CODE                   #
        ###########################################################################
```

```
def dropout_backward(dout, cache):
    """
    Perform the backward pass for (inverted) dropout.

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param['mode']

    dx = None
    if mode == 'train':
        ###########################################################################
        # TODO: Implement training phase backward pass for inverted dropout     #
        ###########################################################################
        dx = mask * dout
        ###########################################################################
        #                                      END OF YOUR CODE                   #
        ###########################################################################
    elif mode == 'test':
        dx = dout
    return dx
```

backward我也连着一起贴上来了

p代表每层神经元的失活概率mode代表是训练模式还是测试模式

Keep_prob是神经元留下来的概率，小于该概率的就直接舍弃，

但是mask*x会使out的均值降低，为了尽量保持原分布，除以keep_prob

In [4]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.3, 0.6, 0.75]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.3
Mean of input:  10.000207878477502
Mean of train-time output:  9.990848162756775
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.300672
Fraction of test-time output set to zero:  0.0
```

```
Running tests with p =  0.6
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.75
Mean of input:  10.000207878477502
Mean of train-time output:  9.991640741515118
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.750232
Fraction of test-time output set to zero:  0.0
```

验证一下结果

In [7]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
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'],
}

solvers = {}
dropout_choices = [0, 0.75]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

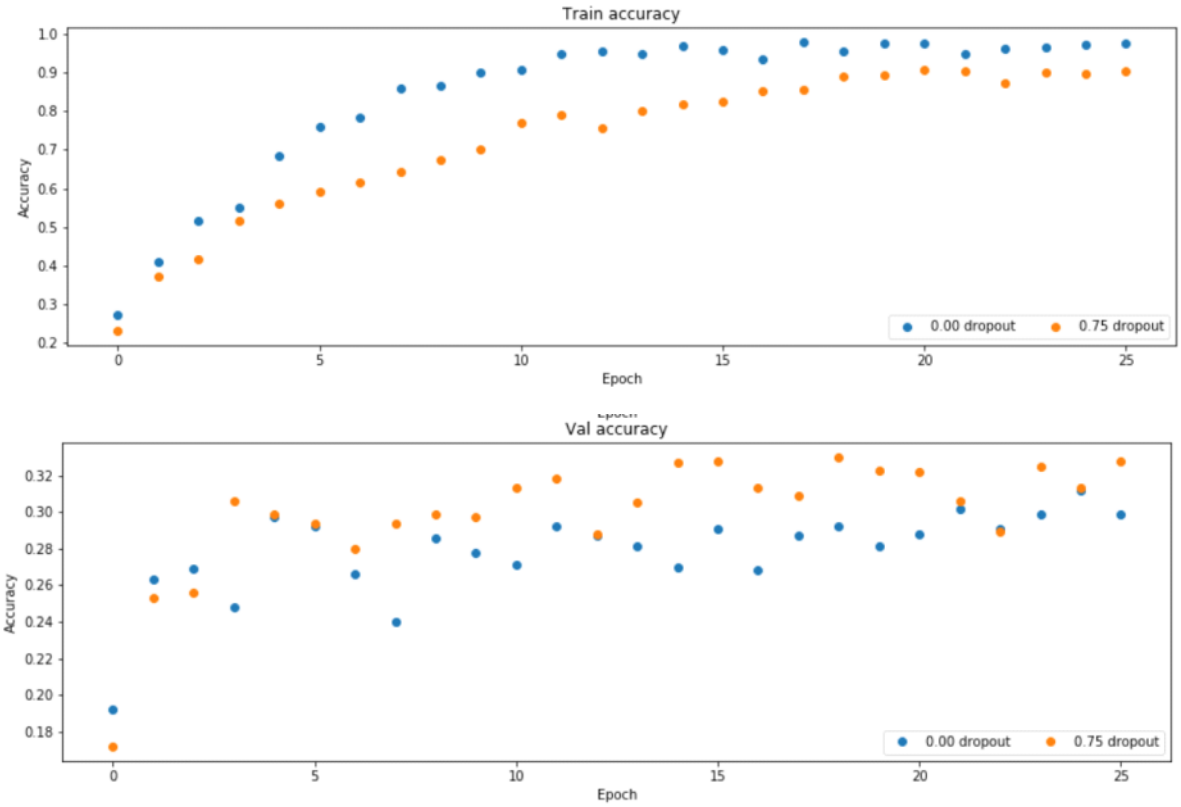
```
0
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.274000; val_acc: 0.192000
(Epoch 1 / 25) train acc: 0.410000; val_acc: 0.263000
(Epoch 2 / 25) train acc: 0.518000; val_acc: 0.269000
(Epoch 3 / 25) train acc: 0.550000; val_acc: 0.248000
(Epoch 4 / 25) train acc: 0.684000; val_acc: 0.297000
(Epoch 5 / 25) train acc: 0.758000; val_acc: 0.292000
(Epoch 6 / 25) train acc: 0.782000; val_acc: 0.266000
(Epoch 7 / 25) train acc: 0.860000; val_acc: 0.240000
(Epoch 8 / 25) train acc: 0.864000; val_acc: 0.286000
```

```
0.75
(Iteration 1 / 125) loss: 17.318480
(Epoch 0 / 25) train acc: 0.232000; val_acc: 0.172000
(Epoch 1 / 25) train acc: 0.372000; val_acc: 0.253000
(Epoch 2 / 25) train acc: 0.416000; val_acc: 0.256000
(Epoch 3 / 25) train acc: 0.516000; val_acc: 0.306000
(Epoch 4 / 25) train acc: 0.560000; val_acc: 0.299000
(Epoch 5 / 25) train acc: 0.592000; val_acc: 0.294000
(Epoch 6 / 25) train acc: 0.614000; val_acc: 0.280000
```

有没有dropout的对比

就截一部分

In [8]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



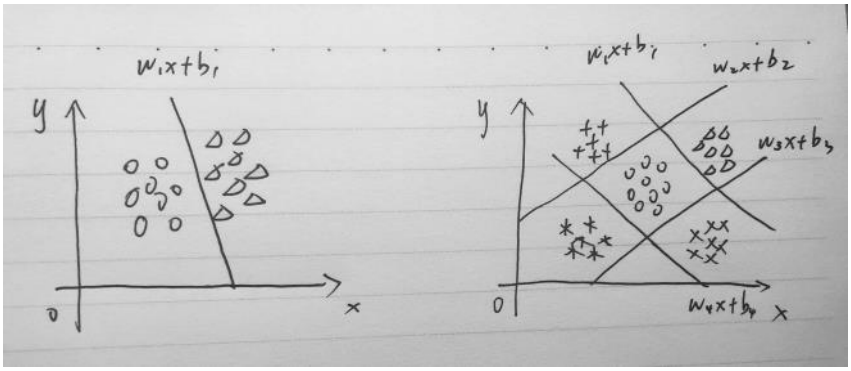
可以看出来用了dropout还是能一定程度上抑制过拟合的

终于终于要开始CNN了！

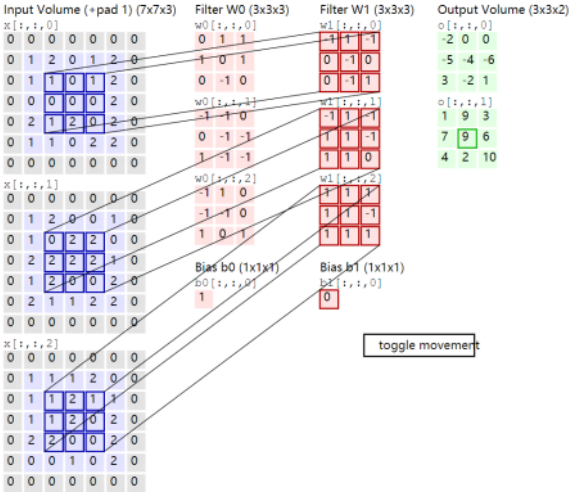
CNN

2019年4月21日 15:44

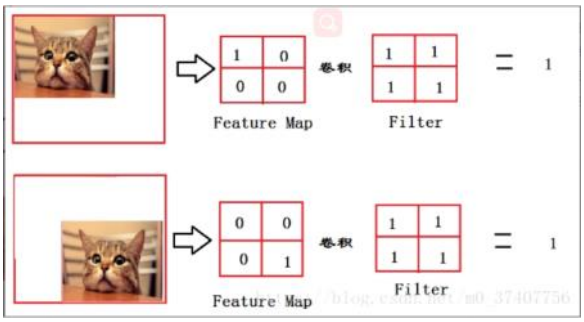
到CNN了，来理一下之前做的东西，以及整个网络各个部分做了些什么东西。
最开始的网络只有一层，将训练样本输入到网络里面，输出一个向量，这个向量的每一个元素代表这个图片在每一个类上所得的分数，然后用一个loss函数来表示这一轮的评估。然后反向求出网络里面参数的梯度，用梯度下降法使得loss函数收敛，最终得到模型。
再往后加入了隐藏层，加入隐藏层的意义是这样的，



看这张图就很好理解了，一条线分割不开就用多条线。
有了全连接层之后分类效果好很多，但是又有了新的问题，如果图片像素点过多，处理起来就很复杂，这时需要对图片进行降维处理。
降维就理解为把一个大的特征分解成几个小特征，比如一只猫，可以把猫分成猫头，猫身子，猫尾巴，然后用一个小窗口对着图片一路扫一遍，找到了这些特征就好做出分类了。
这里放一个图（链接里有动态图）http://cs231n.github.io/convolutional-networks/



这里有一个5*5*3的图片，3表示的是RGB三个通道，然后用了两个filter，每个filter都有三个通道分别在图片的三个通道上面进行扫描，最后输出两个filter的矩阵。对图片进行0填充是为了保持边缘信息，如果不填充边缘数据只会进行一次卷积，还有就是可以保持原始数据维度填充之后可以整除filter维度。
给输入图片做卷积的意义在于，可以消除目标物平移、旋转等扰动对网络的影响，比如：



猫在图片的左上角和右下角卷积之后都为1，因此模型具有更好的鲁棒性。

```
    ###########################################################################
    # TODO: Implement the convolutional forward pass.                         #
    # Hint: you can use the function np.pad for padding.                      #
    ###########################################################################
    N, C, H, W = x.shape
    F, C, HH, WW = w.shape
    stride = conv_param['stride']
    pad = conv_param['pad']

    new_H = 1 + int((H + 2 * pad - HH)/stride)
    new_W = 1 + int((W + 2 * pad - WW)/stride)
    out = np.zeros([N, F, new_H, new_W])

    for n in range(N):
        for f in range(F):
            conv_newH_newW = np.ones([new_H, new_W])*b[f]
            for c in range(C):
                pedded_x = np.lib.pad(x[n, c], pad_width=pad, mode='constant',
                                      constant_values=0)
                for i in range(new_H):
                    for j in range(new_W):
                        conv_newH_newW[i, j]+=np.sum(pedded_x[
                        i*stride:i*stride+HH, j*stride:j*stride+WW]*
                        w[f, c, :, :])
                out[n, f] = conv_newH_newW
    ###########################################################################
    #                                 END OF YOUR CODE                        #
    ###########################################################################
```

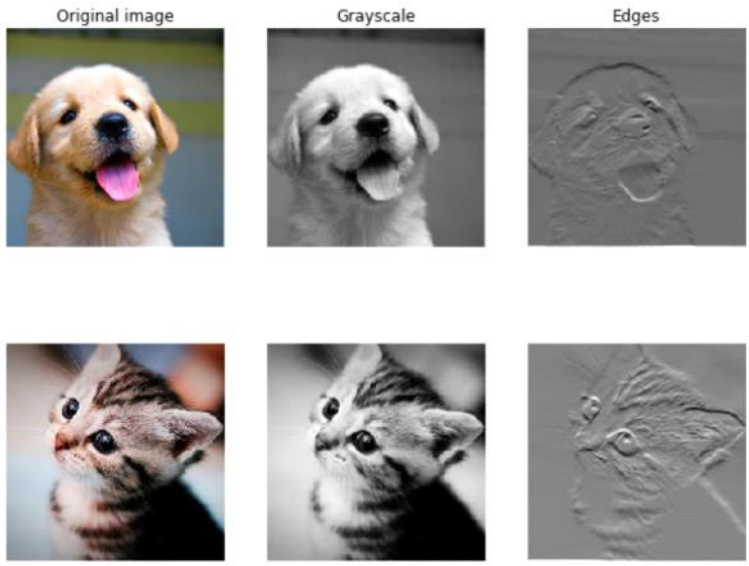
New_H和new_W分别是卷积核在每一行或每一列扫描的次数，也就是最终求完卷积的输出矩阵的shape

```
In [25]: 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 2e-8
         print('Testing conv_forward_naive')
         print('difference: ', rel_error(out, correct_out))

         Testing conv_forward_naive
         difference:  2.2121476417505994e-08
```

验证结果

```
In [4]: 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])
```

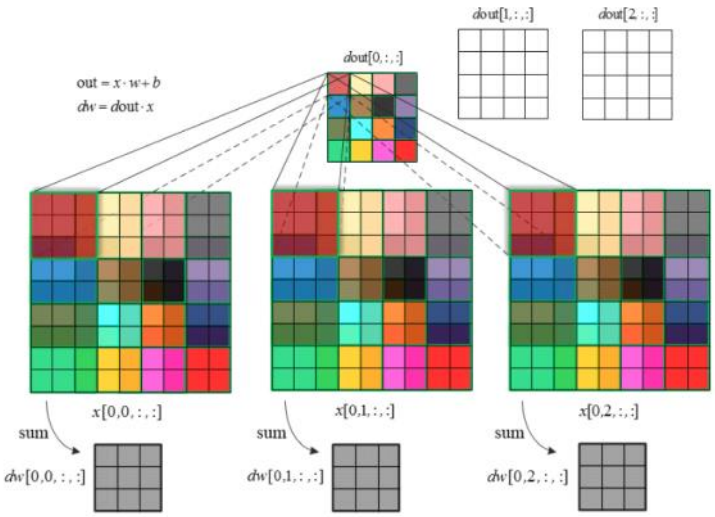
```
# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

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()
```



warning可忽略

下面完成conv的backward

```
###########################################################################
# TODO: Implement the convolutional backward pass.                        #
###########################################################################
x, w, b, conv_param = cache
pad = conv_param['pad']
stride = conv_param['stride']
F, C, HH, WW = w.shape
N, C, H, W = x.shape
N, F, new_H, new_W = dout.shape
padded_x = np.lib.pad(x, ((0,0),(0,0),(pad,pad),(pad,pad)),
                      mode='constant',constant_values=0)
padded_dx = np.zeros_like(padded_x)
dw = np.zeros_like(w)
db = np.zeros_like(b)

for n in range(N):
    for f in range(F):
        for i in range(new_H):
            for j in range(new_W):
                db[f] += dout[n,f,i,j]
                dw[f] += padded_x[n,:,i*stride:HH+i*stride,
                                  j*stride:WW+j*stride]*dout[n,f,i,j]
                padded_dx[n,:,i*stride:HH+i*stride,j*stride:WW+j*stride]\
                +=w[f] * dout[n,f,i,j]
dx = padded_dx[:,:,pad:pad+H,pad:pad+W]
###########################################################################
#                                END OF YOUR CODE                         #
###########################################################################
```



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

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

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

这是反向求导过程

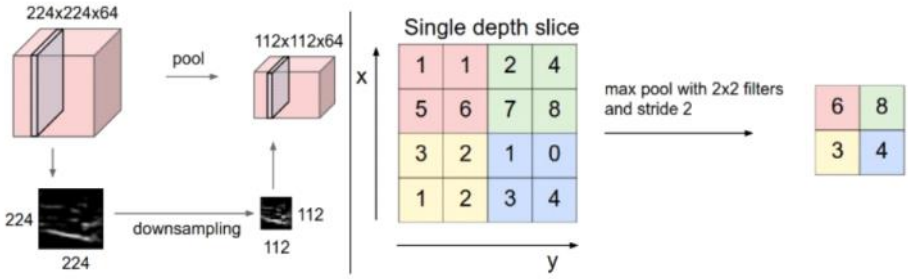
In [8]:
```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

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 1e-8'
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:  4.6975751946308024e-09
dw error:  6.468129276837272e-10
db error:  2.122676399382307e-10
```

验证结果

下面完成池化层的设计

```
###########################################################################
# TODO: Implement the max pooling forward pass
###########################################################################
N, C, H, W = x.shape
pool_height = pool_param['pool_height']
pool_width = pool_param['pool_width']
pool_stride = pool_param['stride']
new_H = 1 + int((H - pool_height)/pool_stride)
new_W = 1 + int((H - pool_width)/pool_stride)

out = np.zeros([N, C, new_H, new_W])
for n in range(N):
      for c in range(C):
            for i in range(new_H):
                  for j in range(new_W):
                        out[n, c, i, j] = np.max(x[n, c, i*pool_stride:i*pool_stride+\
                                                 pool_height, j*pool_stride:j*\
                                                 pool_stride+pool_width])
###########################################################################
#                                         END OF YOUR CODE
###########################################################################
cache = (x, pool_param)
return out, cache


###########################################################################
# TODO: Implement the max pooling backward pass
###########################################################################
x, pool_param = cache
N, C, H, W = x.shape
pool_height = pool_param['pool_height']
pool_width = pool_param['pool_width']
pool_stride = pool_param['stride']
new_H = 1 + int((H - pool_height)/pool_stride)
new_W = 1 + int((W - pool_width)/pool_stride)
dx = np.zeros_like(x)
for n in range(N):
      for c in range(C):
            for i in range(new_H):
                  for j in range(new_W):
                        window = x[n, c, i*pool_stride:i*pool_stride+pool_height,
                                   j*pool_stride:j*pool_stride+pool_width]
                        dx[n, c, i*pool_stride:i*pool_stride+pool_height, j*pool_stride:
                            j*pool_stride+pool_width]=(window==np.max(window))*\
                            dout[n, c, i, j]
###########################################################################
#                                         END OF YOUR CODE
###########################################################################
return dx
```



maxpooling就是选一个窗口里的最大值，反向求导dx就是x的每个窗口里的最大值乘上上游导数，其他值为0

In [9]:
```
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))
```

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

In [7]:
```
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 around 1e-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
```

验证结果

记得assignment2最开始的时候报了一个配置错，是跟cython有关的，在这里第一次用到，这节作业在编写代码时用的基本都是循环嵌套，接下来作业给出了一个用C++实现的更快速的解决方法，连接C++和python需要用到cython库。下面是使用fast_layer的结果

In [8]:
```
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: 14.528142s
Fast: 0.029919s
Speedup: 485.576209x
Difference:  2.7285883131760887e-11

Testing conv_backward_fast:
Naive: 9.570402s
Fast: 0.015957s
Speedup: 599.775502x
dx difference:  1.949764775345631e-11
dw difference:  5.188375174206562e-13
db difference:  3.481354613192702e-14
```

这是卷积层的对比

In [12]:
```
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: 0.403248s
fast: 0.005952s
speedup: 67.751362x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.395745s
fast: 0.024305s
speedup: 57.426556x
dx difference:  0.0
```

这是池化层的对比

Sandwich~

In [11]:
```
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)

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
```

这是包含池化层的

In [14]:
```
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)

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
```

这是不包含池化层的

下面完成cs231n/classifier/cnn.py的threeconvlayer完成一个三层的卷积神经网络

```
############################################################################
# TODO: Initialize weights and biases for the three-layer convolutional    #
# network. Weights should be initialized from a Gaussian with standard     #
# deviation equal to weight_scale; biases should be initialized to zero.   #
# All weights and biases should be stored in the dictionary self.params.   #
# Store weights and biases for the convolutional layer using the keys 'W1' #
# and 'b1'; use keys 'W2' and 'b2' for the weights and biases of the        #
# hidden affine layer, and keys 'W3' and 'b3' for the weights and biases   #
# of the output affine layer.                                              #
############################################################################
C,H,W = input_dim
self.params['W1'] = np.random.normal(0,weight_scale,(num_filters,C,
                                filter_size,filter_size))
self.params['b1'] = np.zeros(num_filters)

self.params['W2'] = np.random.normal(0,weight_scale,(num_filters*\
                                H*W//4,hidden_dim))
#num_filter*H*W//4返回的是池化之后矩阵元素的个数;
self.params['b2'] = np.zeros(hidden_dim)
self.params['W3'] = np.random.normal(0,weight_scale,(hidden_dim,
                                num_classes))
self.params['b3'] = np.zeros(num_classes)
#//表示整数除法，返回不大于结果的一个最大整数，因为是向后兼容，所以前
#面加上from_future_import division
############################################################################
#                                 END OF YOUR CODE                         #
############################################################################
```

初始化参数

```
############################################################################
# TODO: Implement the forward pass for the three-layer convolutional net,  #
# computing the class scores for X and storing them in the scores          #
# variable.                                                                #
############################################################################
crp_out,crp_cache = conv_relu_pool_forward(X,W1,b1,conv_param,pool_param)
crp_out_shape = crp_out.shape
crp_out = crp_out.reshape(crp_out.shape[0],-1)
fc_out,fc_cache = affine_relu_forward(crp_out,W2,b2)
a_out,a_cache = affine_forward(fc_out,W3,b3)
scores = a_out
############################################################################
#                             END OF YOUR CODE                             #
############################################################################
```

```
############################################################################
# TODO: Initialize weights and biases for the three-layer convolutional    #
# network. Weights should be initialized from a Gaussian with standard     #
# deviation equal to weight_scale; biases should be initialized to zero.   #
# All weights and biases should be stored in the dictionary self.params.   #
# Store weights and biases for the convolutional layer using the keys 'W1' #
# and 'b1'; use keys 'W2' and 'b2' for the weights and biases of the       #
# hidden affine layer, and keys 'W3' and 'b3' for the weights and biases   #
# of the output affine layer.                                              #
############################################################################
C,H,W = input_dim
self.params['W1'] = np.random.normal(0,weight_scale,(num_filters,C,
                                filter_size,filter_size))
self.params['b1'] = np.zeros(num_filters)

self.params['W2'] = np.random.normal(0,weight_scale,(num_filters*\
                                H*W//4,hidden_dim))
self.params['b2'] = np.zeros(hidden_dim)
self.params['W3'] = np.random.normal(0,weight_scale,(hidden_dim,
                                num_classes))
self.params['b3'] = np.zeros(num_classes)
#//表示整数除法，返回不大于结果的一个最大整数，因为是向后兼容，所以前
#面加上from_future_import division
############################################################################
#                             END OF YOUR CODE                             #
############################################################################
```

前向传播的步骤

In [16]:
```
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):  2.302585412176879
Initial loss (with regularization):  2.508542482169898
```

比较有正则化和没有正则化的loss

In [14]:
```
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)
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
```

验证结果

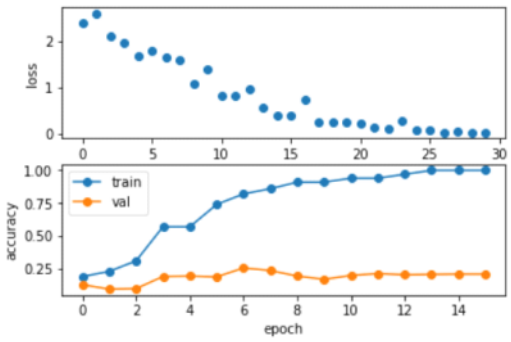
In [15]:
```
np.random.seed(231)

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()
```

用100个数据来训练

```
(Epoch 10 / 15) train acc: 0.940000; val_acc: 0.199000
(Iteration 21 / 30) loss: 0.212628
(Iteration 22 / 30) loss: 0.126721
(Epoch 11 / 15) train acc: 0.940000; val_acc: 0.213000
(Iteration 23 / 30) loss: 0.113127
(Iteration 24 / 30) loss: 0.270010
(Epoch 12 / 15) train acc: 0.970000; val_acc: 0.204000
(Iteration 25 / 30) loss: 0.067624
(Iteration 26 / 30) loss: 0.081088
(Epoch 13 / 15) train acc: 1.000000; val_acc: 0.207000
(Iteration 27 / 30) loss: 0.029606
(Iteration 28 / 30) loss: 0.051089
(Epoch 14 / 15) train acc: 1.000000; val_acc: 0.209000
(Iteration 29 / 30) loss: 0.027549
(Iteration 30 / 30) loss: 0.025578
(Epoch 15 / 15) train acc: 1.000000; val_acc: 0.209000
```

最终得到的结果，截取一部分

In [18]:
```
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()
```



loss和训练集和验证集精度的图片

In [17]:
```
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 701 / 980) loss: 1.447530
(Iteration 721 / 980) loss: 1.538806
(Iteration 741 / 980) loss: 1.893407
(Iteration 761 / 980) loss: 1.408911
(Iteration 781 / 980) loss: 2.160169
(Iteration 801 / 980) loss: 1.850402
(Iteration 821 / 980) loss: 1.542580
(Iteration 841 / 980) loss: 1.441510
(Iteration 861 / 980) loss: 1.779246
(Iteration 881 / 980) loss: 1.656049
(Iteration 901 / 980) loss: 1.455208
(Iteration 921 / 980) loss: 1.820103
(Iteration 941 / 980) loss: 1.526195
(Iteration 961 / 980) loss: 1.639982
(Epoch 1 / 1) train acc: 0.484000; val_acc: 0.485000
```

也是截一部分

In [18]:
```
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()
```

卷积层可视化，可以看到有些相同特征有一些平移旋转的变换

空间批量正则化

这个理念和批量正则化一样，只不过再CNN里面，BN层要做一定的变化

```
################################################################################
# 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
x_new = x.transpose(0,2,3,1).reshape(N*H*W,C)
out,cache = batchnorm_forward(x_new,gamma,beta,bn_param)
out = out.reshape(N,H,W,C).transpose(0,3,1,2)
################################################################################
#                                          END OF YOUR CODE                    #
################################################################################
```

可以看到x先做了一个转置，把H和W提前，把C放在了后面，可以理解为这是在对每个样本的每个通道下求的分布

```
################################################################################
# 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
dout_new = dout.transpose(0,2,3,1).reshape(N*H*W,C)
dx,dgamma,dbeta = batchnorm_backward(dout_new,cache)
dx = dx.reshape(N,H,W,C).transpose(0,3,1,2)
################################################################################
#                                          END OF YOUR CODE                    #
################################################################################
```

同理的反向传播

In [19]:
```
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]
```

In [21]:
```
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]
```

In [22]:
```
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)

_, 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:  7.097481711360870 5e-12
dbeta error:  3.275608725278405e-12
```

验证结果