In [1]:
```
# #COMMENT IF NOT USING COLAB VM
#
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment2/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

In [2]:
```
#UNCOMMENT IF USING CADE
import os
##### Request a GPU #####
## This function locates an available gpu for usage. In addition, this function reserve
## memory space exclusively for your account. The memory reservation prevents the decre
## speed when other users try to allocate memory on the same gpu in the shared systems,
## Note: If you use your own system which has a GPU with less than 4GB of memory, remem
## specified mimimum memory.
def define_gpu_to_use(minimum_memory_mb = 3500):
    thres_memory = 600 #
    gpu_to_use = None
    try:
        os.environ['CUDA_VISIBLE_DEVICES']
        print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE_DEVICES'])
        return
    except:
        pass

    for i in range(16):
        free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=csv,nounits,n
        if free_memory[0] == 'No devices were found':
            break
        free_memory = int(free_memory[0])

        if free_memory>minimum_memory_mb-thres_memory:
            gpu_to_use = i
            break

    if gpu_to_use is None:
        print('Could not find any GPU available with the required free memory of ' + st
              + 'MB. Please use a different system for this assignment.')
    else:
        os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
        print('Chosen GPU: ' + str(gpu_to_use))

## Request a gpu and reserve the memory space
define_gpu_to_use(4000)
```

```
Chosen GPU: 0
```

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In [3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arr
from cs6353.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))))
```

In [4]:
```
# 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,))
```

Affine layer: foward

Open the file `cs6353/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

In [5]:
```
# Test the affine_forward function

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

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
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 e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769848888397517e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

In [6]:
```
# 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)
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 e-10 or less
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:  1.0908199508708189e-10
dw error:  2.1752635504596857e-10
db error:  7.736978834487815e-12
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

In [7]:
```
# 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 on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

In [8]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

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 on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

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

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. Sigmoid
2. ReLU
3. Leaky ReLU

Answer:

1. The activation function with this problem is: sigmoid Relu Leaky Relu
2. Causes: (1) The gradient value of the sigmoid function at its tail end is close to 0. [e^(-10), e^(10)] Such very large negative inputs and very large positive inputs cause the gradient to disappear. (2) This situation occurs in the input range of the Relu function (-∞, 0), such as [-1, -2] (3) Leaky Relu will not have the same situation as Relu when the input is less than 0, but will have a very small gradient (0.01). But when all inputs are 0, [0, 0], Leaky Relu gradient disappears, which may only be due to poor network initialization.

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs6353/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [9]:
```
from cs6353.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, d
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, d
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, d

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
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 and affine_relu_backward:
dx error:  6.395535042049294e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs6353/layers.py`.

You can make sure that the implementations are correct by running the following:

In [10]:
```
np.random.seed(231)
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 around the ord
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 close to 2.3 and dx error should be around
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
```

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs6353/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [11]:
```
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)

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,
    12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,
    12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,
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'

# Errors should be around e-7 or less
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: 1.53e-08
W2 relative error: 3.37e-10
b1 relative error: 8.01e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 7.98e-08
b1 relative error: 1.35e-08
b2 relative error: 1.97e-09
```

Solver

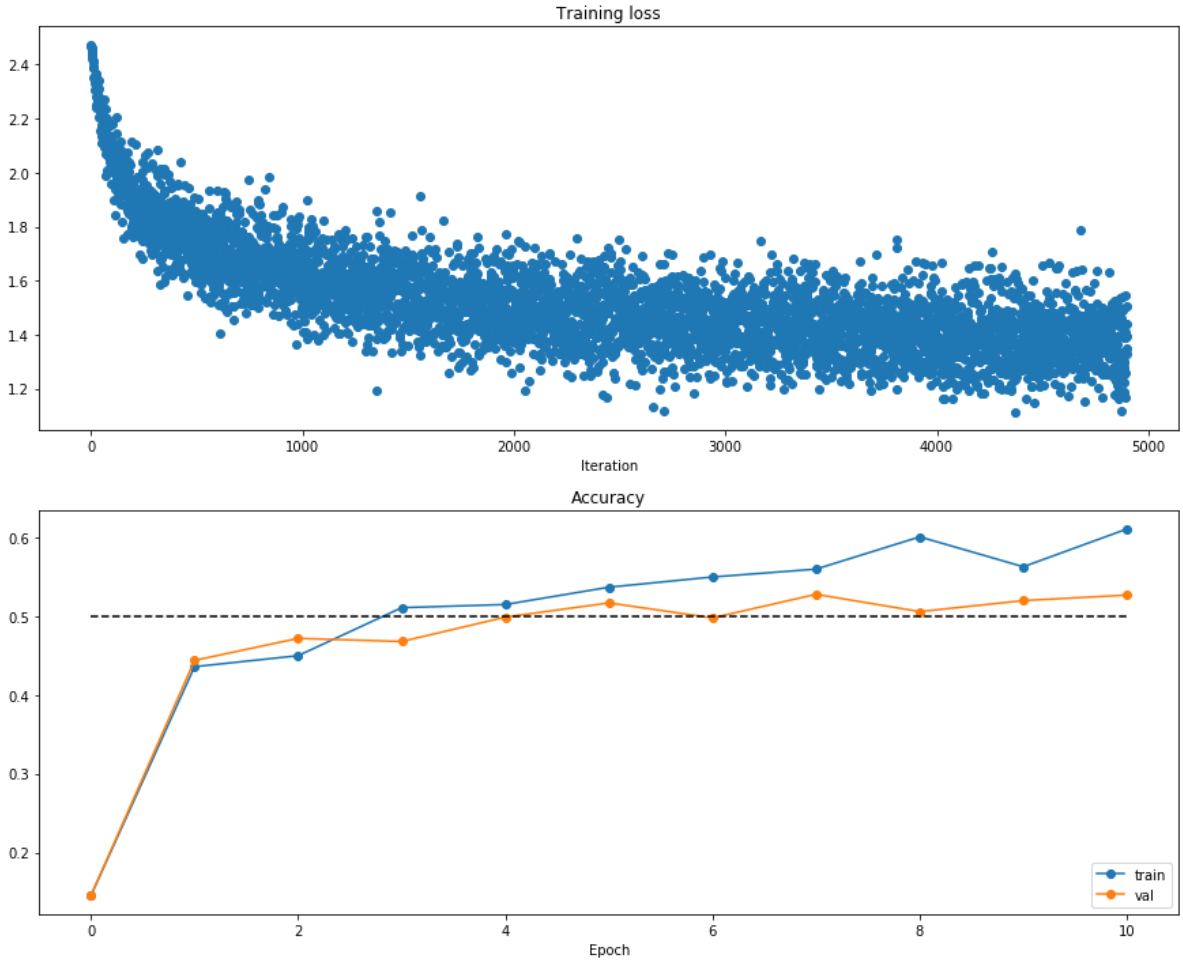
In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Open the file `cs6353/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

In [12]:
```
################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                          #
################################################################################

best_solver = None
best_acc = 0
solver = None

while best_acc <= 0.52:
    solver_hidden_dim = np.random.randint(100, 201)
    solver_reg = np.random.uniform(4, 8) * (10 ** (-1))
    solver_learning_rate = np.random.uniform(1, 9)  * (10 ** np.random.randint(-4, -3))
    print('solver_hidden_dim: ' + str(solver_hidden_dim) + ' solver_reg: ' + str(solver
    model = TwoLayerNet(hidden_dim= solver_hidden_dim, reg= solver_reg)
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': solver_learning_rate,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=10000)
    solver.train()
    if best_acc < solver.best_val_acc:
        best_acc = solver.best_val_acc
        best_solver = solver

print('best acc: ' + str(best_acc))

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

```
solver_hidden_dim: 188 solver_reg: 0.5845246460134131 solver_learning_rate: 0.0005899
695658298456
(Iteration 1 / 4900) loss: 2.472555
(Epoch 0 / 10) train acc: 0.145000; val_acc: 0.145000
(Epoch 1 / 10) train acc: 0.436000; val_acc: 0.444000
(Epoch 2 / 10) train acc: 0.450000; val_acc: 0.472000
(Epoch 3 / 10) train acc: 0.511000; val_acc: 0.468000
(Epoch 4 / 10) train acc: 0.515000; val_acc: 0.499000
(Epoch 5 / 10) train acc: 0.537000; val_acc: 0.517000
(Epoch 6 / 10) train acc: 0.550000; val_acc: 0.498000
(Epoch 7 / 10) train acc: 0.560000; val_acc: 0.528000
(Epoch 8 / 10) train acc: 0.601000; val_acc: 0.506000
(Epoch 9 / 10) train acc: 0.563000; val_acc: 0.520000
(Epoch 10 / 10) train acc: 0.611000; val_acc: 0.527000
best acc: 0.528
```

In [13]:
```
# Run this cell to visualize training loss and train / val accuracy

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

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



Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs6353/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

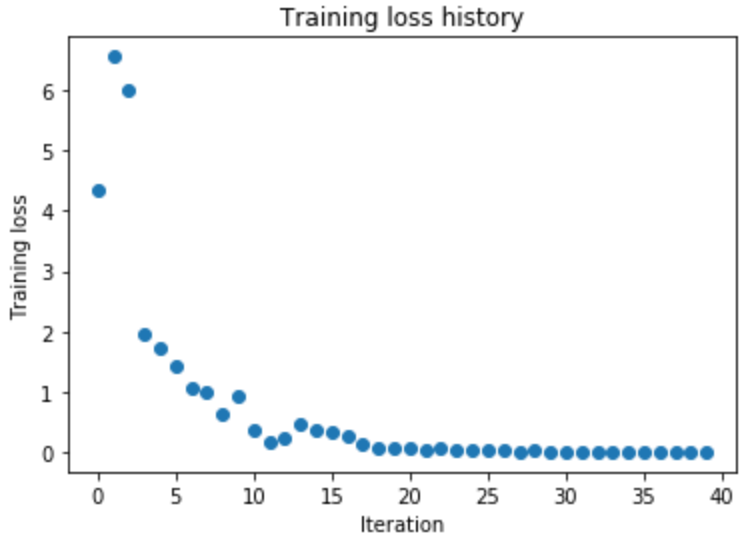
In [14]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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)

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  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
```

As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

In [15]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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 = 2.2e-2
learning_rate = 1e-2
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()
```

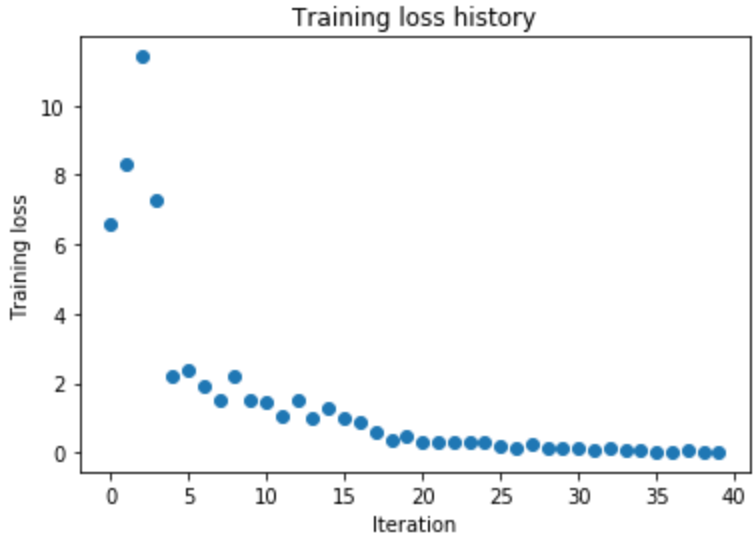
```
(Iteration 1 / 40) loss: 4.349048
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.124000
(Epoch 1 / 20) train acc: 0.340000; val_acc: 0.115000
(Epoch 2 / 20) train acc: 0.600000; val_acc: 0.153000
(Epoch 3 / 20) train acc: 0.620000; val_acc: 0.149000
(Epoch 4 / 20) train acc: 0.780000; val_acc: 0.210000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.187000
(Iteration 11 / 40) loss: 0.367600
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.200000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.205000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.199000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.212000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.185000
(Iteration 21 / 40) loss: 0.080951
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.191000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.191000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.187000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.189000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.191000
(Iteration 31 / 40) loss: 0.019412
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.191000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.185000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.186000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.180000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.181000
```



Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [16]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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 = 2.2e-2
weight_scale = 5.2e-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: 6.601968
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.092000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.108000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.087000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.112000
(Epoch 4 / 20) train acc: 0.540000; val_acc: 0.130000
(Epoch 5 / 20) train acc: 0.560000; val_acc: 0.129000
(Iteration 11 / 40) loss: 1.457960
(Epoch 6 / 20) train acc: 0.700000; val_acc: 0.131000
(Epoch 7 / 20) train acc: 0.620000; val_acc: 0.138000
(Epoch 8 / 20) train acc: 0.780000; val_acc: 0.128000
(Epoch 9 / 20) train acc: 0.900000; val_acc: 0.142000
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.161000
(Iteration 21 / 40) loss: 0.296019
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.143000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.158000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.154000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.141000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.146000
(Iteration 31 / 40) loss: 0.150492
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.138000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.154000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.139000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.140000
```



Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer:

1. Training a five-layer neural network is more difficult than a three-layer neural network
2. The five-layer neural network is more sensitive to data initialization
3. This is because the five-layer neural network is deeper than the three-layer neural network and is more susceptible to data initialization. Data initialization that is too small will cause more gradients to disappear, and data initialization that is too large will cause more gradients to explode.

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent.

Open the file `cs6353/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

In [17]:
```
from cs6353.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    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

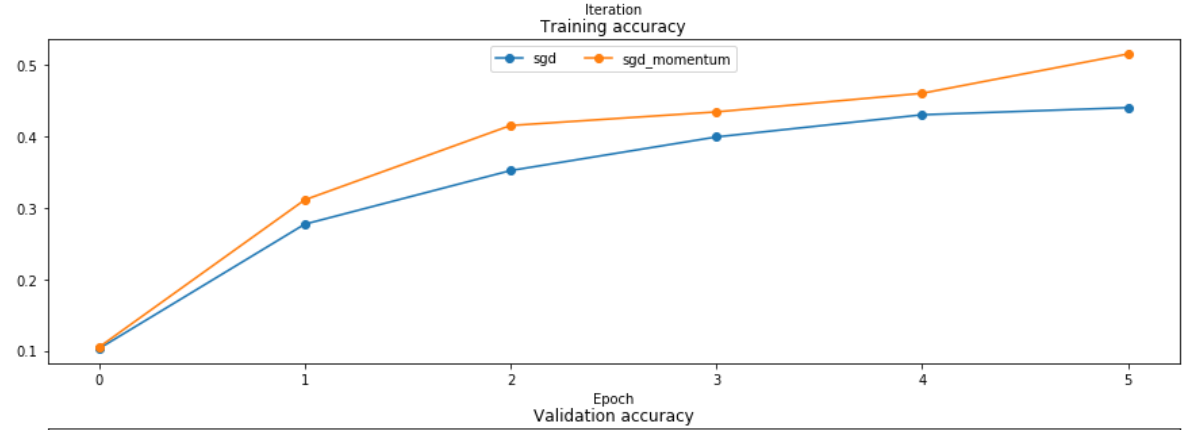
```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

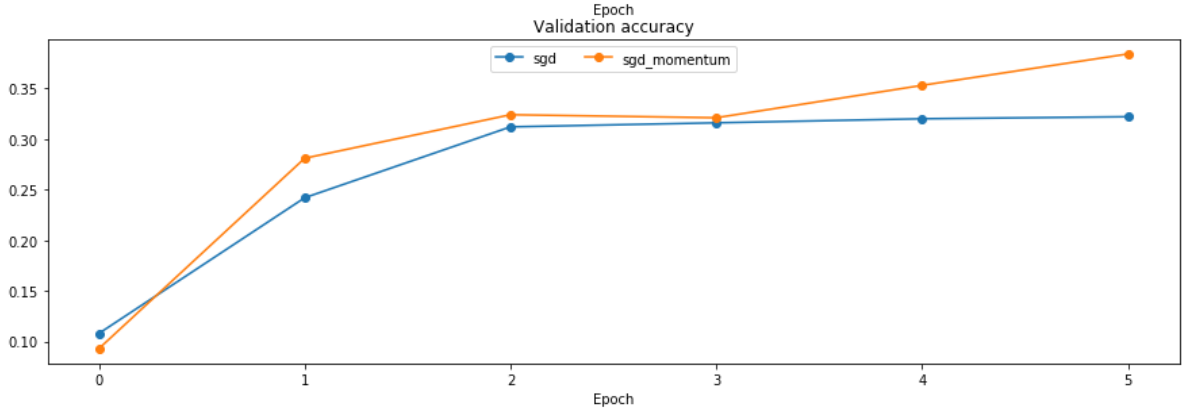
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

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

```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082693
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.023753
(Iteration 71 / 200) loss: 2.026621
(Epoch 2 / 5) train acc: 0.352000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.807163
(Iteration 91 / 200) loss: 1.914256
(Iteration 101 / 200) loss: 1.920494
(Iteration 111 / 200) loss: 1.708877
(Epoch 3 / 5) train acc: 0.399000; val_acc: 0.316000
(Iteration 121 / 200) loss: 1.701111
(Iteration 131 / 200) loss: 1.769697
(Iteration 141 / 200) loss: 1.788898
(Iteration 151 / 200) loss: 1.815921
(Epoch 4 / 5) train acc: 0.430000; val_acc: 0.320000
(Iteration 161 / 200) loss: 1.631982
(Iteration 171 / 200) loss: 1.896426
(Iteration 181 / 200) loss: 1.545108
(Iteration 191 / 200) loss: 1.713720
(Epoch 5 / 5) train acc: 0.440000; val_acc: 0.322000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032562
(Iteration 31 / 200) loss: 1.985848
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882354
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512047
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610417
(Iteration 181 / 200) loss: 1.528331
(Iteration 191 / 200) loss: 1.447238
(Epoch 5 / 5) train acc: 0.515000; val_acc: 0.384000
```

```
C:\Users\Nitocris\AppData\Roaming\Python\Python36\site-packages\ipykernel_launcher.p
y:39: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a prev
ious axes currently reuses the earlier instance.  In a future version, a new instance
will always be created and returned.  Meanwhile, this warning can be suppressed, and
the future behavior ensured, by passing a unique label to each axes instance.
C:\Users\Nitocris\AppData\Roaming\Python\Python36\site-packages\ipykernel_launcher.p
y:42: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a prev
ious axes currently reuses the earlier instance.  In a future version, a new instance
will always be created and returned.  Meanwhile, this warning can be suppressed, and
the future behavior ensured, by passing a unique label to each axes instance.
C:\Users\Nitocris\AppData\Roaming\Python\Python36\site-packages\ipykernel_launcher.p
y:45: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a prev
ious axes currently reuses the earlier instance.  In a future version, a new instance
will always be created and returned.  Meanwhile, this warning can be suppressed, and
the future behavior ensured, by passing a unique label to each axes instance.
C:\Users\Nitocris\AppData\Roaming\Python\Python36\site-packages\ipykernel_launcher.p
y:49: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a prev
ious axes currently reuses the earlier instance.  In a future version, a new instance
will always be created and returned.  Meanwhile, this warning can be suppressed, and
the future behavior ensured, by passing a unique label to each axes instance.
```





Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

In [19]:
```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################

best_solver = None
best_acc = 0
solver = None

while best_acc <= 0.51:
    solver_reg = np.random.uniform(1, 8) * (10 ** (-2))
    solver_learning_rate = np.random.uniform(1, 9)  * (10 ** np.random.randint(-4, -3))
    dropout = np.random.uniform(6, 8) * (10 ** (-1))
    epochs = np.random.randint(10, 21)
    print(' solver_reg: ' + str(solver_reg) + ' solver_learning_rate: ' + str(solver_l
    model = FullyConnectedNet([100, 100, 100, 100, 100], reg= solver_reg, dropout= drop
    solver = Solver(model, data,
                    update_rule='sgd_momentum',
                    optim_config={
                      'learning_rate': solver_learning_rate,
                    },
                    lr_decay=0.95,
                    num_epochs=epochs, batch_size=200,
                    print_every=10000)
    solver.train()
    if best_acc < solver.best_val_acc:
        best_acc = solver.best_val_acc
        best_solver = solver
        best_model = model

print('best acc: ' + str(best_acc))

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

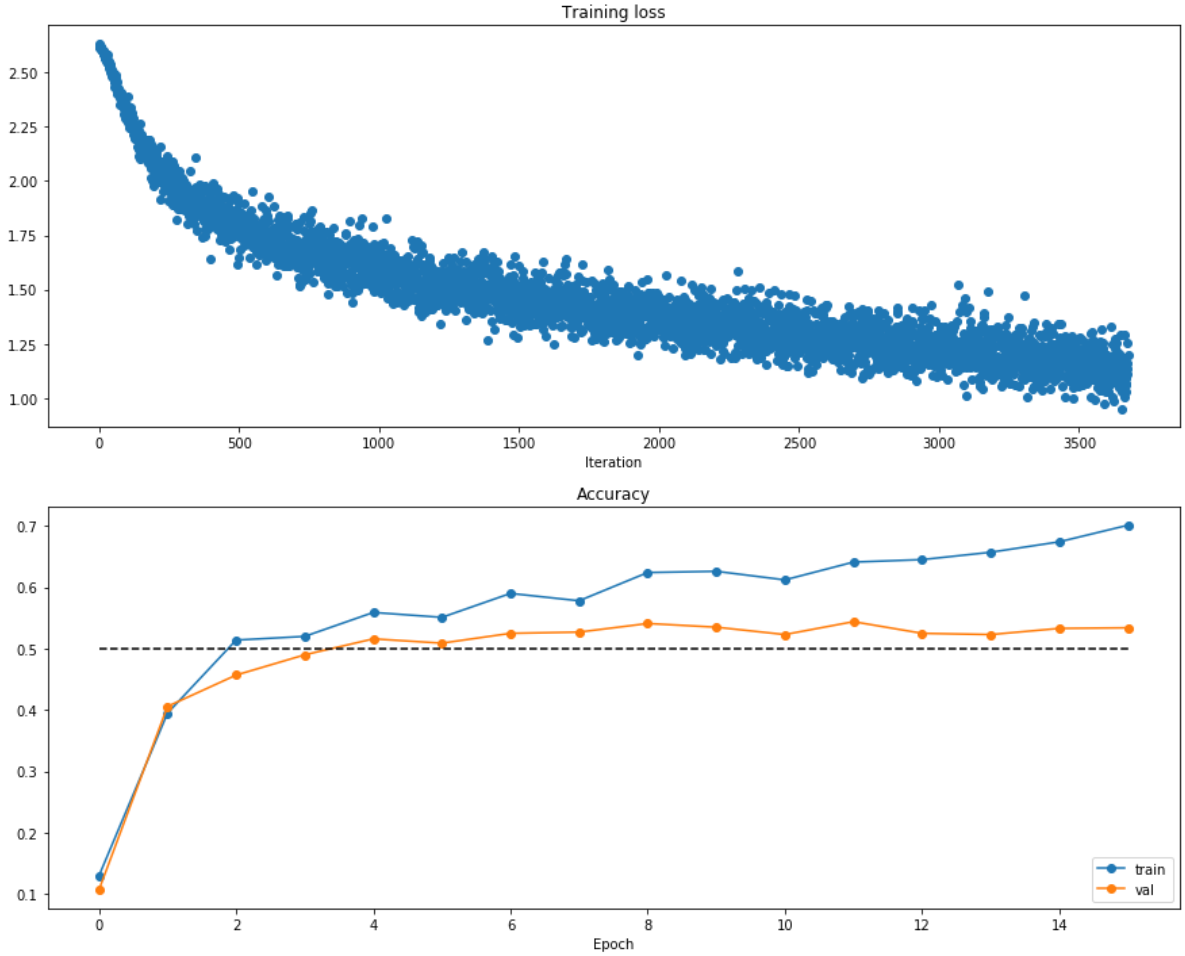
```
 solver_reg: 0.01831920225912328 solver_learning_rate: 0.0007944894378317135 dropout
is : 0.7374521923170966 epoch : 15
(Iteration 1 / 3675) loss: 2.617299
(Epoch 0 / 15) train acc: 0.130000; val_acc: 0.107000
(Epoch 1 / 15) train acc: 0.395000; val_acc: 0.406000
(Epoch 2 / 15) train acc: 0.514000; val_acc: 0.457000
(Epoch 3 / 15) train acc: 0.520000; val_acc: 0.490000
(Epoch 4 / 15) train acc: 0.559000; val_acc: 0.516000
(Epoch 5 / 15) train acc: 0.551000; val_acc: 0.509000
(Epoch 6 / 15) train acc: 0.590000; val_acc: 0.525000
(Epoch 7 / 15) train acc: 0.578000; val_acc: 0.527000
(Epoch 8 / 15) train acc: 0.624000; val_acc: 0.541000
(Epoch 9 / 15) train acc: 0.626000; val_acc: 0.535000
(Epoch 10 / 15) train acc: 0.612000; val_acc: 0.523000
(Epoch 11 / 15) train acc: 0.641000; val_acc: 0.544000
(Epoch 12 / 15) train acc: 0.645000; val_acc: 0.525000
(Epoch 13 / 15) train acc: 0.657000; val_acc: 0.523000
(Epoch 14 / 15) train acc: 0.674000; val_acc: 0.533000
(Epoch 15 / 15) train acc: 0.701000; val_acc: 0.534000
best acc: 0.544
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

In [20]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())

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

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

```
Validation set accuracy:  0.552
Test set accuracy:  0.534
```



In [ ]:

In [1]:
```
# #COMMENT IF NOT USING COLAB VM
#
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment2/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

In [2]:
```
#UNCOMMENT IF USING CADE
import os
##### Request a GPU #####
## This function locates an available gpu for usage. In addition, this function reserve
## memory space exclusively for your account. The memory reservation prevents the decre
## speed when other users try to allocate memory on the same gpu in the shared systems,
## Note: If you use your own system which has a GPU with less than 4GB of memory, remem
## specified mimimum memory.
def define_gpu_to_use(minimum_memory_mb = 3500):
    thres_memory = 600 #
    gpu_to_use = None
    try:
        os.environ['CUDA_VISIBLE_DEVICES']
        print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE_DEVICES'])
        return
    except:
        pass

    for i in range(16):
        free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=csv,nounits,n
        if free_memory[0] == 'No devices were found':
            break
        free_memory = int(free_memory[0])

        if free_memory>minimum_memory_mb-thres_memory:
            gpu_to_use = i
            break

    if gpu_to_use is None:
        print('Could not find any GPU available with the required free memory of ' + st
              + 'MB. Please use a different system for this assignment.')
    else:
        os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
        print('Chosen GPU: ' + str(gpu_to_use))

## Request a gpu and reserve the memory space
define_gpu_to_use(4000)
```

```
Chosen GPU: 0
```

Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization which was proposed by [3] in 2015.

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [3] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

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

In [3]:
```
# As usual, a bit of setup
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arr
from cs6353.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'

%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))))

def print_mean_std(x,axis=0):
    print('  means: ', x.mean(axis=axis))
    print('  stds:  ', x.std(axis=axis))
    print()
```

In [4]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

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

Batch Normalization: Forward

In the file `cs6353/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above would be helpful!

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_mean_std(a,axis=0)

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

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])
# Now means should be close to beta and stds close to gamma
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,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)
  means:  [ 1.33226763e-17 -3.94129174e-17  3.29597460e-17]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  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_mean_std(a_norm,axis=0)
```

```
After batch normalization (test-time):
  means:  [ 2.44249065e-17  5.59274849e-17 -9.54791801e-17]
  stds:   [1. 1. 1.]
```

Batch normalization: Backward Pass

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Once you have finished, run the following to numerically check your backward pass.

In [7]:
```
# 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)
#You should expect to see relative errors between 1e-13 and 1e-8
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:  1.7029303759326446e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

Batch Normalization: Alternative Backward

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can do a similar simplification for the batch normalization backward pass too.

Given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix}$, we first calculate the mean $\mu = \frac{1}{N}\sum_{k=1}^{N} x_k$ and variance $v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2$.
With μ and v calculated, we can calculate the standard deviation $\sigma = \sqrt{v + \epsilon}$ and normalized data Y with $y_i = \frac{x_i - \mu}{\sigma}$.

The meat of our problem is to get $\frac{\partial L}{\partial X}$ from the upstream gradient $\frac{\partial L}{\partial Y}$. It might be challenging to directly reason about the gradients over X and Y - try reasoning about it in terms of x_i and y_i first.

You will need to come up with the derivations for $\frac{\partial L}{\partial x_i}$, by relying on the Chain Rule to first calculate the intermediate $\frac{\partial \mu}{\partial x_i}$, $\frac{\partial v}{\partial x_i}$, $\frac{\partial \sigma}{\partial x_i}$, then assemble these pieces to calculate $\frac{\partial y_i}{\partial x_i}$. You should make sure each of the intermediary steps are all as simple as possible.

After doing so, implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

In [8]:
```
np.random.seed(231)
N, D = 100, 500
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'}
out, cache = batchnorm_forward(x, gamma, beta, bn_param)

t1 = time.time()
dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
t2 = time.time()
dx2, dgamma2, dbeta2 = batchnorm_backward_alt(dout, cache)
t3 = time.time()

print('dx difference: ', rel_error(dx1, dx2))
print('dgamma difference: ', rel_error(dgamma1, dgamma2))
print('dbeta difference: ', rel_error(dbeta1, dbeta2))
print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
```

```
dx difference:  2.606679051793896e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 4.00x
```

Fully Connected Networks with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs6353/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

Concretely, when the `normalization` flag is set to `"batchnorm"` in the constructor, you should insert a batch normalization layer before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized. Once you are done, run the following to gradient-check your implementation.

HINT: You might find it useful to define an additional helper layer similar to those in the file `cs6353/layer_utils.py`.

In [9]:
```
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,))

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
# and losses between 1e-08~1e-09 for beta and gammas.
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,
                            normalization='batchnorm')

  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: 2.22e-03
b2 relative error: 3.55e-07
b3 relative error: 4.78e-11
beta1 relative error: 7.33e-09
beta2 relative error: 1.07e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 2.41e-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: 1.78e-07
b2 relative error: 1.42e-06
b3 relative error: 2.64e-10
beta1 relative error: 6.65e-09
beta2 relative error: 4.23e-09
gamma1 relative error: 5.94e-09
gamma2 relative error: 4.14e-09
```

Batch Normalization for Deep Networks

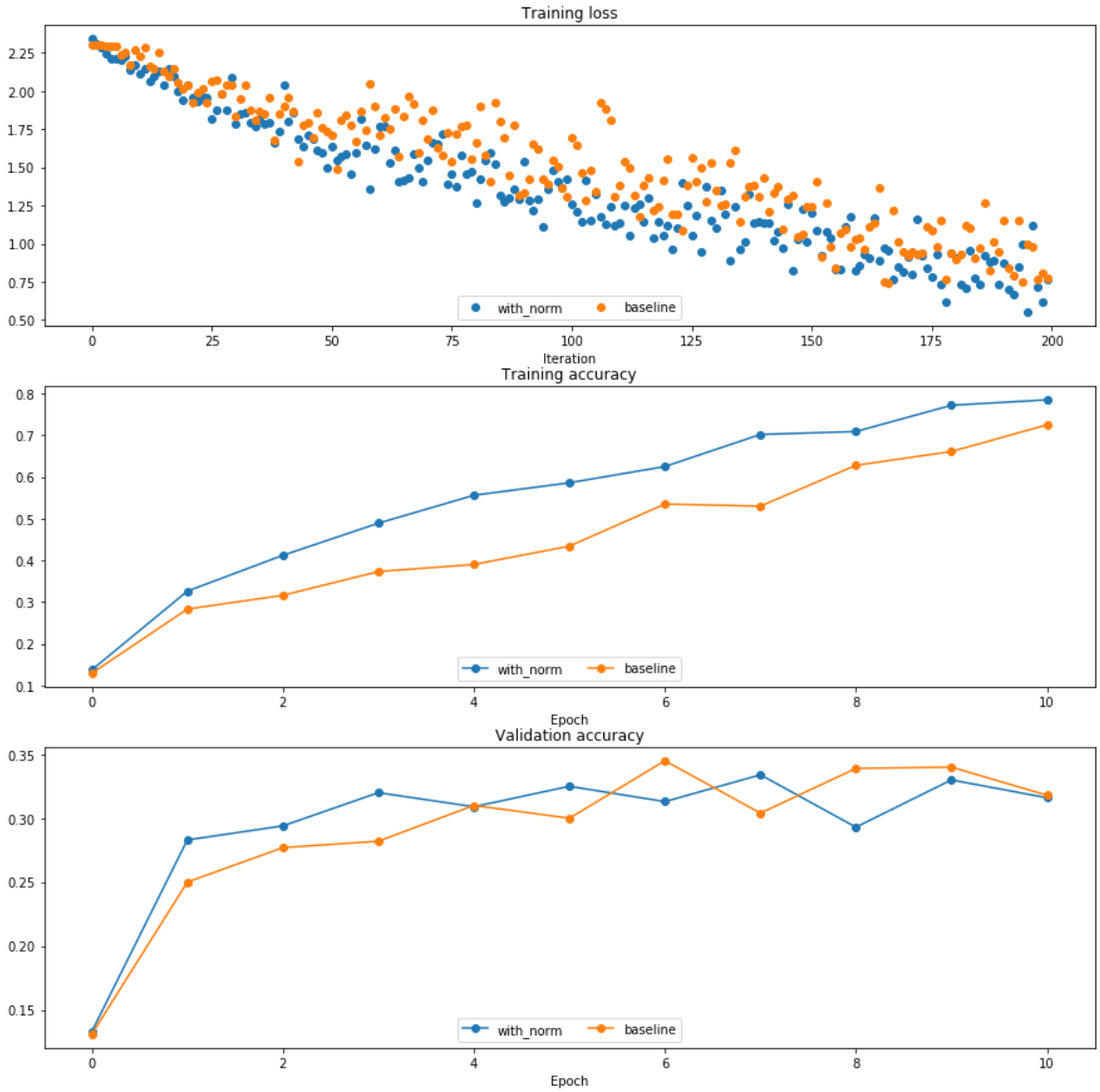
Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

In [10]:
```
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, normalization='bat
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=None)

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=20)
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=20)
solver.train()
```

```
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.137000; val_acc: 0.133000
(Epoch 1 / 10) train acc: 0.326000; val_acc: 0.283000
(Iteration 21 / 200) loss: 2.039365
(Epoch 2 / 10) train acc: 0.412000; val_acc: 0.294000
(Iteration 41 / 200) loss: 2.036710
(Epoch 3 / 10) train acc: 0.489000; val_acc: 0.320000
(Iteration 61 / 200) loss: 1.769536
(Epoch 4 / 10) train acc: 0.556000; val_acc: 0.309000
(Iteration 81 / 200) loss: 1.265761
(Epoch 5 / 10) train acc: 0.586000; val_acc: 0.325000
(Iteration 101 / 200) loss: 1.256780
(Epoch 6 / 10) train acc: 0.625000; val_acc: 0.313000
(Iteration 121 / 200) loss: 1.115818
(Epoch 7 / 10) train acc: 0.702000; val_acc: 0.334000
(Iteration 141 / 200) loss: 1.139977
(Epoch 8 / 10) train acc: 0.709000; val_acc: 0.293000
(Iteration 161 / 200) loss: 0.857926
(Epoch 9 / 10) train acc: 0.772000; val_acc: 0.330000
(Iteration 181 / 200) loss: 0.907038
(Epoch 10 / 10) train acc: 0.785000; val_acc: 0.316000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.131000
(Epoch 1 / 10) train acc: 0.283000; val_acc: 0.250000
(Iteration 21 / 200) loss: 2.041970
(Epoch 2 / 10) train acc: 0.316000; val_acc: 0.277000
(Iteration 41 / 200) loss: 1.900473
(Epoch 3 / 10) train acc: 0.373000; val_acc: 0.282000
(Iteration 61 / 200) loss: 1.713156
(Epoch 4 / 10) train acc: 0.390000; val_acc: 0.310000
(Iteration 81 / 200) loss: 1.662209
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696059
(Epoch 6 / 10) train acc: 0.535000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.557986
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.304000
(Iteration 141 / 200) loss: 1.432189
(Epoch 8 / 10) train acc: 0.628000; val_acc: 0.339000
(Iteration 161 / 200) loss: 1.033932
(Epoch 9 / 10) train acc: 0.661000; val_acc: 0.340000
(Iteration 181 / 200) loss: 0.901033
(Epoch 10 / 10) train acc: 0.725000; val_acc: 0.318000
```

Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

In [11]:
```
def plot_training_history(title, label, baseline, bn_solvers, plot_fn, bl_marker='.', b
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.train_acc_history, bl_marker='-o', bn_marker='-o')
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.val_acc_history, bl_marker='-o', bn_marker='-o')

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



Batch Normalization and Initialization

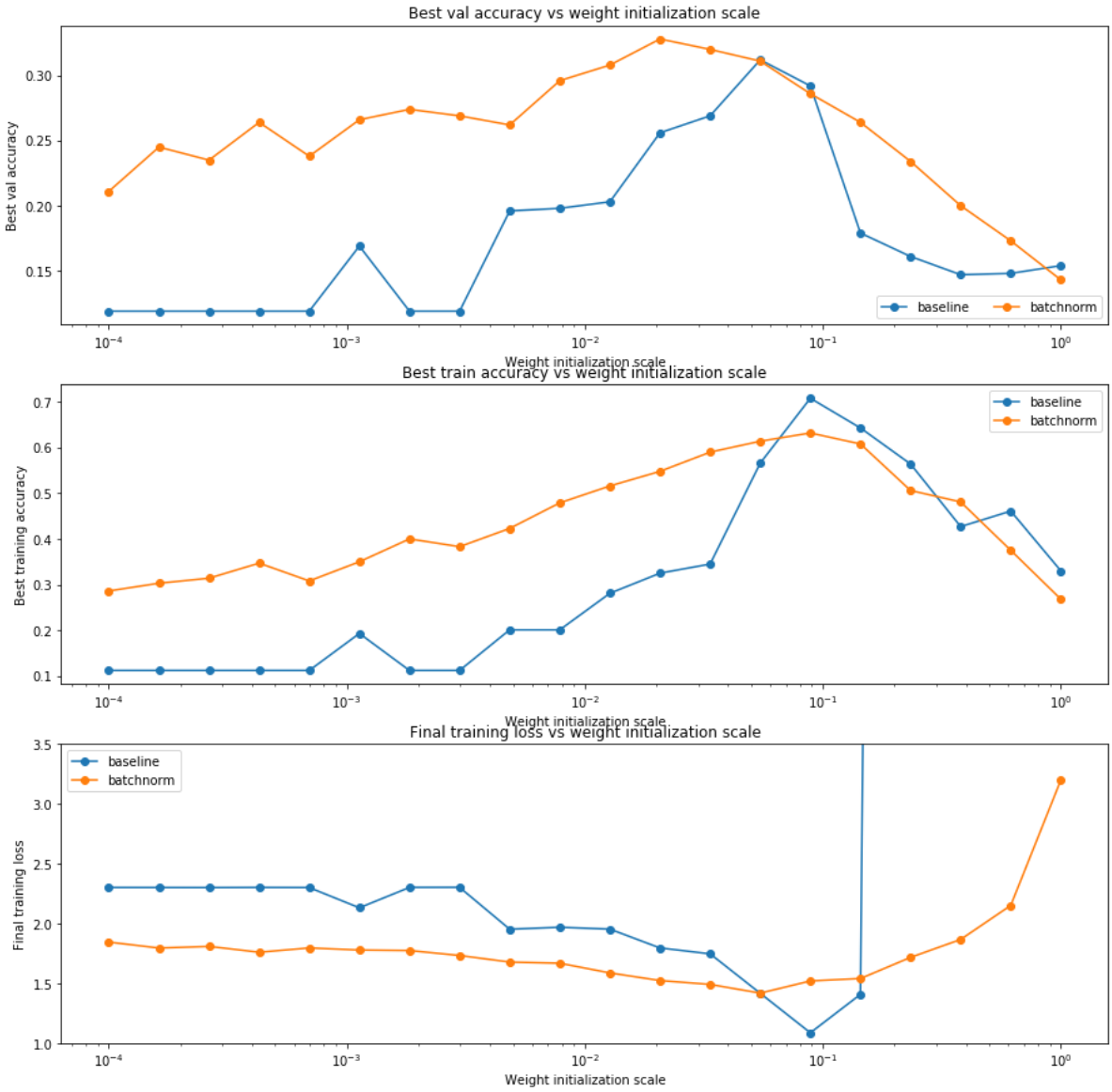
We will now run a small experiment to study the interaction of batch normalization and weight initialization.

The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

In [12]:
```
np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
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'],
}

bn_solvers_ws = {}
solvers_ws = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
  bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization='b
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=None)

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

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

```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

In [13]:
```
# Plot results of weight scale experiment
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

for ws in weight_scales:
  best_train_accs.append(max(solvers_ws[ws].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers_ws[ws].train_acc_history))

  best_val_accs.append(max(solvers_ws[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers_ws[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers_ws[ws].loss_history[-100:]))

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()
plt.gca().set_ylim(1.0, 3.5)

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



Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

Answer:

1. Training models with batch normalization generally perform better than models without batch normalization.
2. Overfitting can be alleviated by using batch normalization.
3. Too small and too large weight scale will have less impact on the model using batch normalization.
4. Reason:
   A. Batch normalization has a regularization effect
   B. Batch normalization can reduce the impact of gradient disappearance caused by too small weight initialization and gradient explosion caused by too large weight initialization because batch normalization causes each data point to disperse from the batch mean by 1 on average.

Batch normalization and batch size

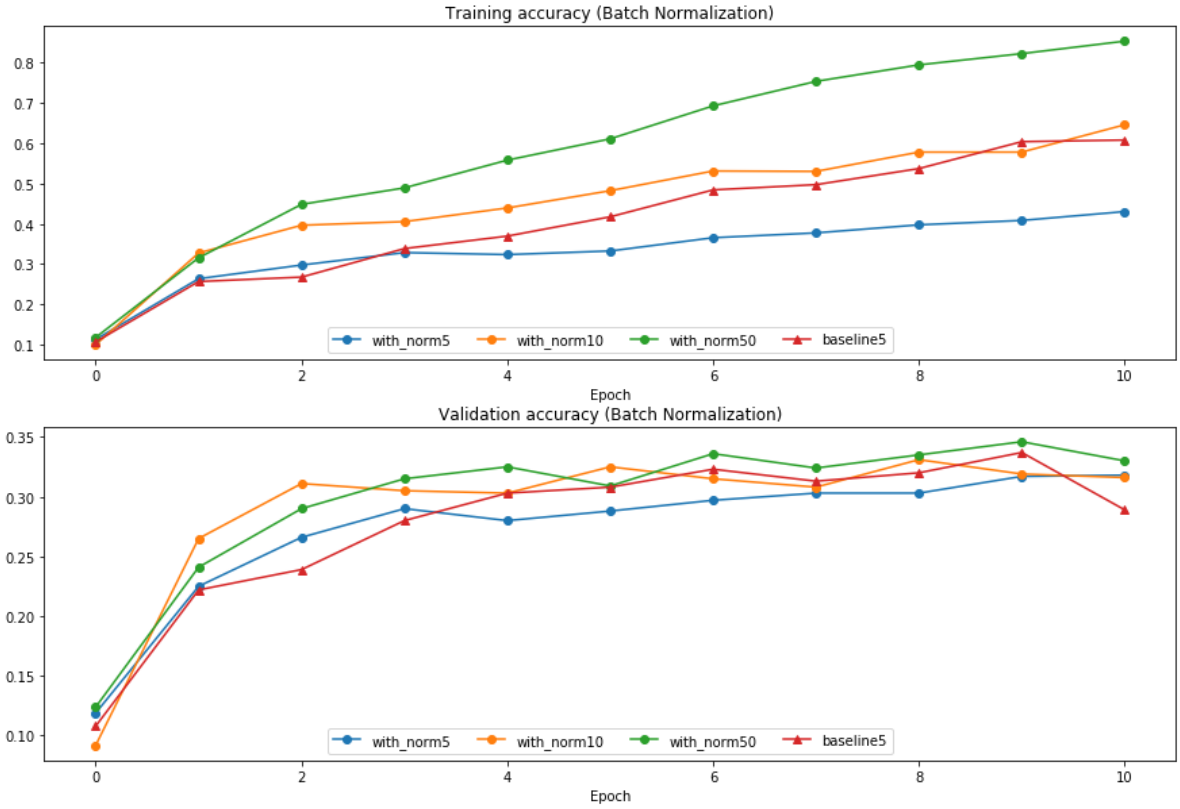
We will now run a small experiment to study the interaction of batch normalization and batch size.

The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

In [14]:
```
def run_batchsize_experiments(normalization_mode):
    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'],
    }
    n_epochs=10
    weight_scale = 2e-2
    batch_sizes = [5,10,50]
    lr = 10**(-3.5)
    solver_bsize = batch_sizes[0]

    print('No normalization: batch size = ',solver_bsize)
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=Nor
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': lr,
                    },
                    verbose=False)
    solver.train()

    bn_solvers = []
    for i in range(len(batch_sizes)):
        b_size=batch_sizes[i]
        print('Normalization: batch size = ',b_size)
        bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalizat
        bn_solver = Solver(bn_model, small_data,
                        num_epochs=n_epochs, batch_size=b_size,
                        update_rule='adam',
                        optim_config={
                          'learning_rate': lr,
                        },
                        verbose=False)
        bn_solver.train()
        bn_solvers.append(bn_solver)

    return bn_solvers, solver, batch_sizes

batch_sizes = [5,10,50]
bn_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('batchnorm')
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```

In [15]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch', solver_bsize, l
                      lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch', solver_bsize,
                      lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lab

plt.gcf().set_size_inches(15, 10)
plt.show()
```



Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

Answer:

1. Using batch normalization on a larger batch size will improve training accuracy more than using it on a smaller batch size.
2. Using batch normalization on a very small batch size is not as good as the base line without batch normalization in terms of training set accuracy.
3. Using batch normalization on a larger batch size will improve the accuracy of the verification set more than on a smaller batch size, because it can accelerate convergence. However, as the epoch increases, other situations eventually converge, so the accuracy of the final verification set is not improved well.
4. The reason why batch normalization does not work well on small batch sizes is that when the batch is too small, calculating the mean and variance and performing normalization on it will make the data noisy. By increasing the batch size, a better approximation will be found.

Layer Normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks which have a cap on the input batch size due to hardware limitations.

Several alternatives to batch normalization have been proposed to mitigate this problem; one such technique is Layer Normalization [2]. Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

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

Inline Question 3:

Which of these data preprocessing steps is analogous to batch normalization, and which is analogous to layer normalization?

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.
3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

Answer:

1. 3 is analogous to batch normalization
2. 2 is analogous to layer normalization
   A. batch normalization: Each feature is normalized based on same feature acorss multiple samples
   B. layer normalization: Each feature is normalized based on other freatures across the same samples

Layer Normalization: Implementation

Now you'll implement layer normalization. This step should be relatively straightforward, as conceptually the implementation is almost identical to that of batch normalization. One significant difference though is that for layer normalization, we do not keep track of the moving moments, and the testing phase is identical to the training phase, where the mean and variance are directly calculated per datapoint.

Here's what you need to do:

- In `cs6353/layers.py`, implement the forward pass for layer normalization in the function `layernorm_backward`.

Run the cell below to check your results.

- In `cs6353/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results.

- Modify `cs6353/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNet`. When the `normalization` flag is set to `"layernorm"` in the constructor, you should insert a layer normalization layer before each ReLU nonlinearity.

Run the third cell below to run the batch size experiment on layer normalization.

In [16]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after layer normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 =4, 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 layer normalization:')
print_mean_std(a,axis=1)

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

gamma = np.asarray([3.0,3.0,3.0])
beta = np.asarray([5.0,5.0,5.0])
# Now means should be close to beta and stds close to gamma
print('After layer normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)
```

```
Before layer normalization:
  means:  [-59.06673243 -47.60782686 -43.31137368 -26.40991744]
  stds:   [10.07429373 28.39478981 35.28360729  4.01831507]

After layer normalization (gamma=1, beta=0)
  means:  [ 4.81096644e-16  0.00000000e+00  0.00000000e+00 -2.96059473e-16]
  stds:   [0.99999995 0.99999999 1.         0.99999969]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means:  [5. 5. 5. 5.]
  stds:   [2.99999985 2.99999998 2.99999999 2.99999907]
```

In [17]:
```
# 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)

ln_param = {}
fx = lambda x: layernorm_forward(x, gamma, beta, ln_param)[0]
fg = lambda a: layernorm_forward(x, a, beta, ln_param)[0]
fb = lambda b: layernorm_forward(x, gamma, b, ln_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 = layernorm_forward(x, gamma, beta, ln_param)
dx, dgamma, dbeta = layernorm_backward(dout, cache)

#You should expect to see relative errors between 1e-12 and 1e-8
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

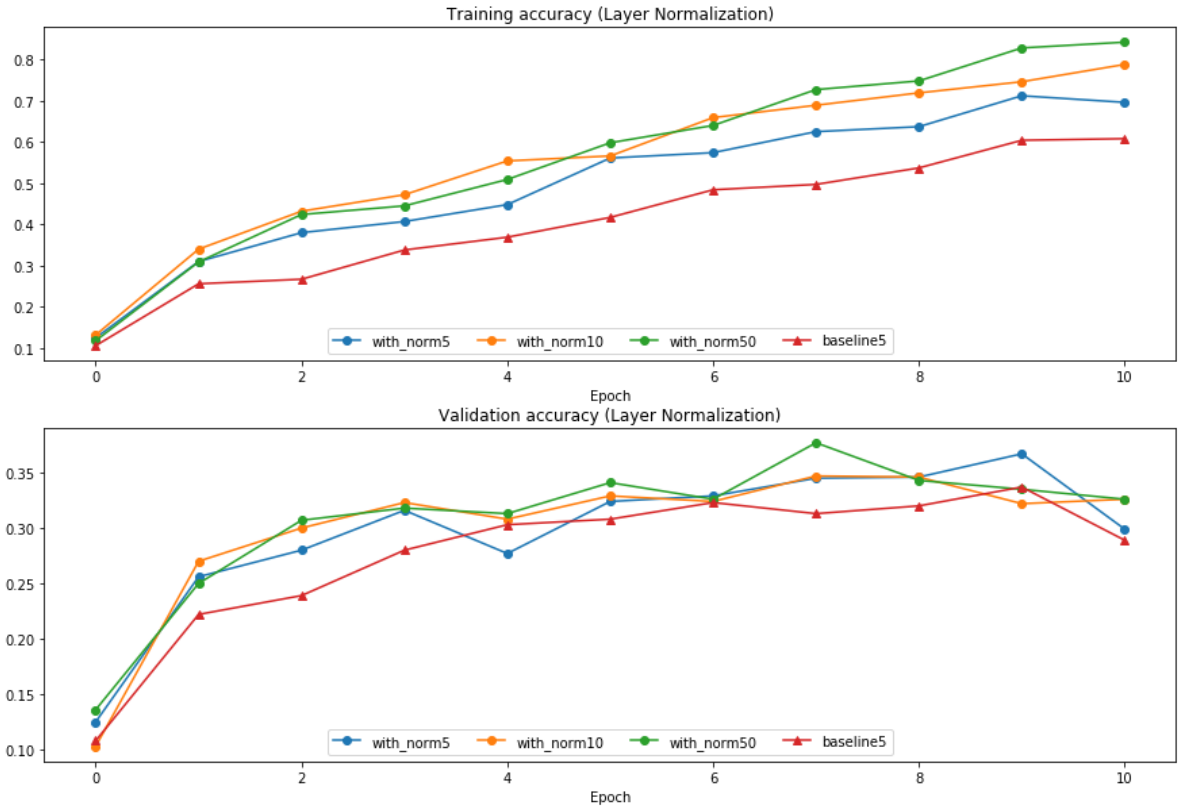
```
dx error:  2.107279147162234e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.5842537629899423e-12
```

Layer Normalization and batch size

We will now run the previous batch size experiment with layer normalization instead of batch normalization. Compared to the previous experiment, you should see a markedly smaller influence of batch size on the training history!

In [18]:
```
ln_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('layernorm')

plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Layer Normalization)','Epoch', solver_bsize,
                      lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Layer Normalization)','Epoch', solver_bsize,
                      lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lab

plt.gcf().set_size_inches(15, 10)
plt.show()
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```



Inline Question 4:

When is layer normalization likely to not work well, and why?

1. Using it in a very deep network
2. Having a very small dimension of features
3. Having a high regularization term

Answer:

1. Layer normalization does not work well in cases 2 and 3
2. Layer normalization normalizes by layer, so more hidden layers will not reduce its performance
3. Very small feature dimensions may cause noise after layer normalization.

4. A high degree of regularization will cause a large penalty to the weight and increase the loss, so it will not be possible to perform weighting well

In [ ]:

In [1]:
```
# #COMMENT IF NOT USING COLAB
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cs6353/assignments/assignment2/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# # This downloads the CIFAR-10 dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

In [2]:
```
#UNCOMMENT IF USING CADE
import os
##### Request a GPU #####
## This function locates an available gpu for usage. In addition, this function reserve
## memory space exclusively for your account. The memory reservation prevents the decre
## speed when other users try to allocate memory on the same gpu in the shared systems,
## Note: If you use your own system which has a GPU with less than 4GB of memory, remem
## specified mimimum memory.
def define_gpu_to_use(minimum_memory_mb = 3500):
    thres_memory = 600 #
    gpu_to_use = None
    try:
        os.environ['CUDA_VISIBLE_DEVICES']
        print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE_DEVICES'])
        return
    except:
        pass

    for i in range(16):
        free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=csv,nounits,n
        if free_memory[0] == 'No devices were found':
            break
        free_memory = int(free_memory[0])

        if free_memory>minimum_memory_mb-thres_memory:
            gpu_to_use = i
            break

    if gpu_to_use is None:
        print('Could not find any GPU available with the required free memory of ' + st
              + 'MB. Please use a different system for this assignment.')
    else:
        os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
        print('Chosen GPU: ' + str(gpu_to_use))

## Request a gpu and reserve the memory space
define_gpu_to_use(4000)
```

```
Chosen GPU: 0
```

Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012 (https://arxiv.org/abs/1207.0580)

In [3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arr
from cs6353.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))))
```

In [4]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

Dropout forward pass

In the file `cs6353/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

In [5]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  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 =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
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 =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `cs6353/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

In [6]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param)[0

# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

Answer:

1. The expected value will change after passing dropout
2. E[y] = py + (1-p)0 If not /p, E[y] = py
3. If /p, E[y] = py/p = y, no change

Fully-connected nets with Dropout

In the file `cs6353/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

In [7]:
```
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 dropout in [1, 0.75, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

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

  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  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])))
  print()
```

```
Running check with dropout =  1
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 dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.90e-07
W2 relative error: 4.76e-06
W3 relative error: 2.60e-08
b1 relative error: 4.73e-09
b2 relative error: 1.82e-09
b3 relative error: 1.70e-10

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 5.37e-09
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

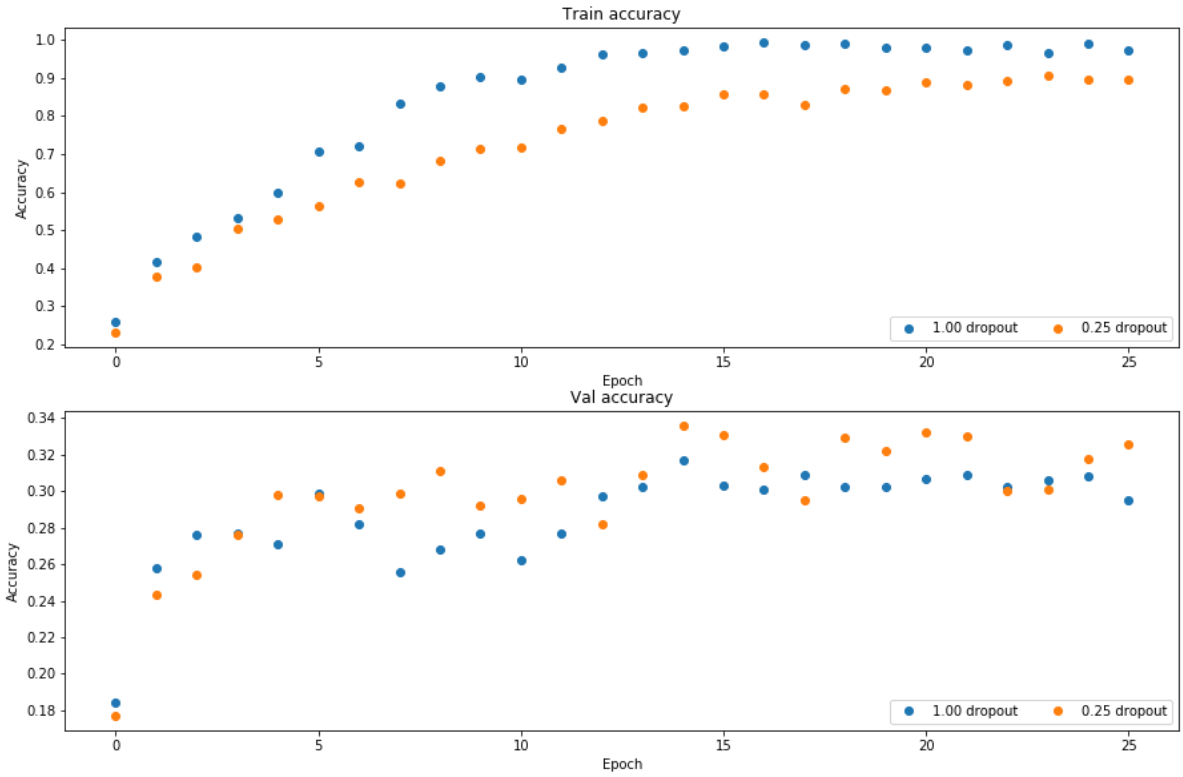
Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

In [8]:
```
# 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 = [1, 0.25]
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
```

```
1
(Iteration 1 / 125) loss: 7.856644
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.256000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.896000; val_acc: 0.262000
(Epoch 11 / 25) train acc: 0.928000; val_acc: 0.277000
(Epoch 12 / 25) train acc: 0.962000; val_acc: 0.297000
(Epoch 13 / 25) train acc: 0.966000; val_acc: 0.302000
(Epoch 14 / 25) train acc: 0.974000; val_acc: 0.317000
(Epoch 15 / 25) train acc: 0.984000; val_acc: 0.303000
(Epoch 16 / 25) train acc: 0.994000; val_acc: 0.301000
(Epoch 17 / 25) train acc: 0.986000; val_acc: 0.309000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.302000
(Epoch 19 / 25) train acc: 0.982000; val_acc: 0.302000
(Epoch 20 / 25) train acc: 0.982000; val_acc: 0.307000
(Iteration 101 / 125) loss: 0.132043
(Epoch 21 / 25) train acc: 0.972000; val_acc: 0.309000
(Epoch 22 / 25) train acc: 0.986000; val_acc: 0.302000
(Epoch 23 / 25) train acc: 0.966000; val_acc: 0.306000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.308000
(Epoch 25 / 25) train acc: 0.974000; val_acc: 0.295000
0.25
(Iteration 1 / 125) loss: 17.318480
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.299000
(Epoch 8 / 25) train acc: 0.684000; val_acc: 0.311000
(Epoch 9 / 25) train acc: 0.714000; val_acc: 0.292000
(Epoch 10 / 25) train acc: 0.718000; val_acc: 0.296000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.306000
(Epoch 12 / 25) train acc: 0.786000; val_acc: 0.282000
(Epoch 13 / 25) train acc: 0.822000; val_acc: 0.309000
(Epoch 14 / 25) train acc: 0.826000; val_acc: 0.336000
(Epoch 15 / 25) train acc: 0.858000; val_acc: 0.331000
(Epoch 16 / 25) train acc: 0.856000; val_acc: 0.313000
(Epoch 17 / 25) train acc: 0.830000; val_acc: 0.295000
(Epoch 18 / 25) train acc: 0.872000; val_acc: 0.329000
(Epoch 19 / 25) train acc: 0.868000; val_acc: 0.322000
(Epoch 20 / 25) train acc: 0.890000; val_acc: 0.332000
(Iteration 101 / 125) loss: 4.364418
(Epoch 21 / 25) train acc: 0.882000; val_acc: 0.330000
(Epoch 22 / 25) train acc: 0.892000; val_acc: 0.300000
(Epoch 23 / 25) train acc: 0.906000; val_acc: 0.301000
(Epoch 24 / 25) train acc: 0.896000; val_acc: 0.318000
(Epoch 25 / 25) train acc: 0.896000; val_acc: 0.326000
```

In [9]:
```
# 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()
```



Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

1. Using dropout as a regularizer is very good
2. The accuracy of the model using dropout on the training set is lower than that of the model not used, but the accuracy on the verification set is similar to or better than the model not used, effectively preventing overfitting.

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

1. If you want to reduce network complexity, you do not need to change p.
2. If you want to make the network complexity similar or unchanged before reducing the hidden layer, you need to increase p.

In [ ]:

#Comment this block if you are NOT using colabVM

This mounts your Google Drive to the Colab VM.

from google.colab import drive drive.mount('/content/drive')

TODO: Enter the foldername in your Drive where you have saved the unzipped

assignment folder, e.g. 'cs6353/assignments/assignment1/'

FOLDERNAME = None assert FOLDERNAME is not None, "[!] Enter the foldername."

Now that we've mounted your Drive, this ensures that

the Python interpreter of the Colab VM can load

python files from within it.

import sys sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

This downloads the CIFAR-10 dataset to your Drive

if it doesn't already exist.

%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/!bash get_datasets.sh
FOLDERNAME

In [1]:
```
#UNCOMMENT IF USING CADE
import os
##### Request a GPU #####
## This function locates an available gpu for usage. In addition, this function reserve
## memory space exclusively for your account. The memory reservation prevents the decre
## speed when other users try to allocate memory on the same gpu in the shared systems,
## Note: If you use your own system which has a GPU with less than 4GB of memory, remem
## specified mimimum memory.
def define_gpu_to_use(minimum_memory_mb = 3500):
    thres_memory = 600 #
    gpu_to_use = None
    try:
        os.environ['CUDA_VISIBLE_DEVICES']
        print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE_DEVICES'])
        return
    except:
        pass

    for i in range(16):
        free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=csv,nounits,n
        if free_memory[0] == 'No devices were found':
            break
        free_memory = int(free_memory[0])

        if free_memory>minimum_memory_mb-thres_memory:
            gpu_to_use = i
            break

    if gpu_to_use is None:
        print('Could not find any GPU available with the required free memory of ' + st
              + 'MB. Please use a different system for this assignment.')
    else:
        os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
        print('Chosen GPU: ' + str(gpu_to_use))

## Request a gpu and reserve the memory space
define_gpu_to_use(4000)
```

```
Chosen GPU: 0
```

Convolutional Networks

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

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

In [2]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.cnn import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient_array, eval_numerical_gradie
from cs6353.layers import *
from cs6353.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))))
```

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

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

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

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs6353/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

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

You can test your implementation by running the following:

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

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

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

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

Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs6353/layers.py`. Again, you don't need to worry too much about computational efficiency.

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

In [5]:
```
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_para
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_para
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_para

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

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

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

Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs6353/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

In [6]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)


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

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

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

Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs6353/layers.py`. You don't need to worry about computational efficiency.

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

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)

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

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

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

Spatial Batch Normalization

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

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

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

[3] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network

Spatial batch normalization: forward

In the file `cs6353/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

In [8]:
```
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 [9]:
```
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:  [-7.69687344e-15 -6.68606584e-15 -5.13021864e-15 -1.09345193e-14]
  stds:  [0.99999898 0.9999991  0.99999909 0.99999904]
```

Spatial batch normalization: backward

In the file `cs6353/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

In [10]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

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

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

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

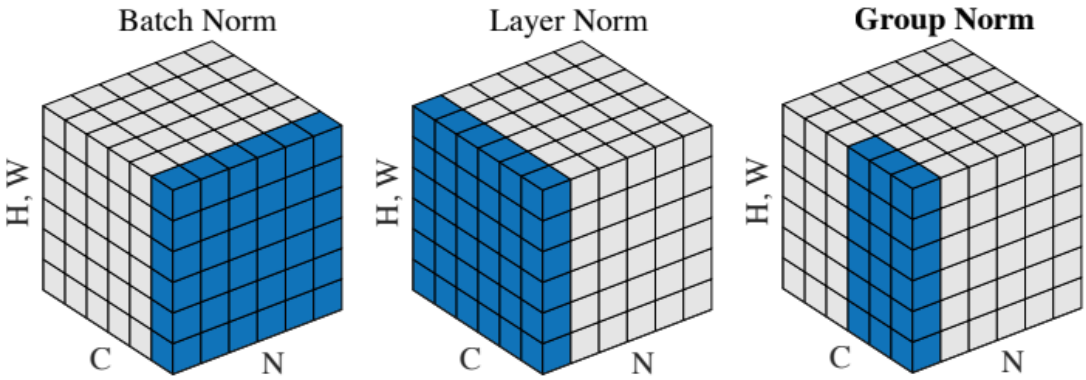
```
dx error:  2.786648201640115e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

Group Normalization

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

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

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



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

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

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

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050

Spatial Group normalization: forward

In the file `cs6353/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

In [11]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

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

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

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

```
Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  Shape:  (1, 1, 1, 2, 6, 4, 5)
  Means:  [-2.14643118e-16  5.25505565e-16  2.58126853e-16 -3.62672855e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

Spatial group normalization: backward

In the file `cs6353/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

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

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

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

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

```
dx error:  6.34590431845254e-08
dgamma error:  1.0546047434202244e-11
dbeta error:  3.810857316122484e-12
```

ConvNet

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

Open the file `cs6353/classifiers/cnn.py` and complete the implementation of the `ConvNet` class. Run the following cells to help you debug:

Sanity check loss

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

In [13]:
```
# num_inputs = 2
# input_dim = (3, 16, 16)
# reg = 0.0
# num_classes = 10
# np.random.seed(231)
# X = np.random.randn(num_inputs, *input_dim)
# y = np.random.randint(num_classes, size=num_inputs)

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


model = ConvNet()

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

# print(model.loss(X, y))

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)
print('log(10): ', np.log(10))

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

```
Initial loss (no regularization):  2.3116596580092312
log(10):  2.302585092994046
Initial loss (with regularization):  2.3356916237697454
```

Gradient check

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

In [14]:
```
num_inputs = 2
input_dim = (3, 8, 8)
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 = ConvNet(input_dim=input_dim, dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=Fals
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
    # print('%s max relative error: %e' % (param_name, param_grad_num))
```

```
W_1 max relative error: 4.684024e-01
W_2 max relative error: 3.027209e-05
W_3 max relative error: 1.521006e-06
b_1 max relative error: 1.628472e-08
b_2 max relative error: 1.613293e-08
b_3 max relative error: 9.360861e-10
beta1 max relative error: 1.034391e-06
beta2 max relative error: 4.716913e-07
gamma1 max relative error: 3.499269e-06
gamma2 max relative error: 1.537313e-05
```

Overfit small data

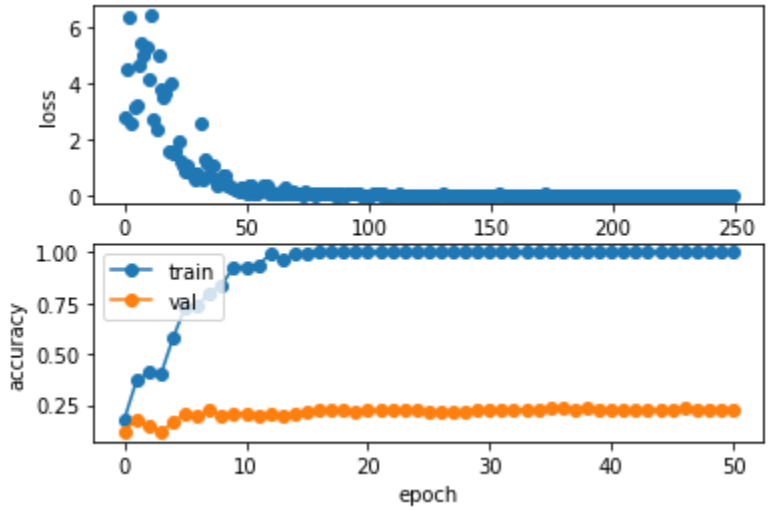
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in high training accuracy and comparatively low validation accuracy.

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 = ConvNet(
    num_filters=[16, 32],
    filter_sizes=[7, 3],
    weight_scale=1e-2
)

solver = Solver(
    model, small_data,
    num_epochs=50, batch_size=20,
    update_rule='sgd_momentum',
    optim_config={
      'learning_rate': 1e-2,
    },
    verbose=True, print_every=10
)
solver.train()
```

```
(Iteration 1 / 250) loss: 2.735618
(Epoch 0 / 50) train acc: 0.180000; val_acc: 0.121000
(Epoch 1 / 50) train acc: 0.370000; val_acc: 0.181000
(Epoch 2 / 50) train acc: 0.410000; val_acc: 0.146000
(Iteration 11 / 250) loss: 4.138183
(Epoch 3 / 50) train acc: 0.400000; val_acc: 0.115000
(Epoch 4 / 50) train acc: 0.580000; val_acc: 0.166000
(Iteration 21 / 250) loss: 1.486943
(Epoch 5 / 50) train acc: 0.730000; val_acc: 0.209000
(Epoch 6 / 50) train acc: 0.740000; val_acc: 0.194000
(Iteration 31 / 250) loss: 0.785646
(Epoch 7 / 50) train acc: 0.800000; val_acc: 0.224000
(Epoch 8 / 50) train acc: 0.840000; val_acc: 0.200000
(Iteration 41 / 250) loss: 0.729946
(Epoch 9 / 50) train acc: 0.930000; val_acc: 0.206000
(Epoch 10 / 50) train acc: 0.930000; val_acc: 0.211000
(Iteration 51 / 250) loss: 0.055936
(Epoch 11 / 50) train acc: 0.940000; val_acc: 0.201000
(Epoch 12 / 50) train acc: 0.990000; val_acc: 0.208000
(Iteration 61 / 250) loss: 0.105686
(Epoch 13 / 50) train acc: 0.970000; val_acc: 0.192000
(Epoch 14 / 50) train acc: 0.990000; val_acc: 0.208000
(Iteration 71 / 250) loss: 0.043670
(Epoch 15 / 50) train acc: 0.990000; val_acc: 0.213000
(Epoch 16 / 50) train acc: 1.000000; val_acc: 0.226000
(Iteration 81 / 250) loss: 0.021199
(Epoch 17 / 50) train acc: 1.000000; val_acc: 0.231000
(Epoch 18 / 50) train acc: 1.000000; val_acc: 0.223000
(Iteration 91 / 250) loss: 0.007575
(Epoch 19 / 50) train acc: 1.000000; val_acc: 0.217000
(Epoch 20 / 50) train acc: 1.000000; val_acc: 0.223000
(Iteration 101 / 250) loss: 0.007665
(Epoch 21 / 50) train acc: 1.000000; val_acc: 0.224000
(Epoch 22 / 50) train acc: 1.000000; val_acc: 0.223000
(Iteration 111 / 250) loss: 0.009133
(Epoch 23 / 50) train acc: 1.000000; val_acc: 0.222000
(Epoch 24 / 50) train acc: 1.000000; val_acc: 0.222000
(Iteration 121 / 250) loss: 0.002350
(Epoch 25 / 50) train acc: 1.000000; val_acc: 0.216000
(Epoch 26 / 50) train acc: 1.000000; val_acc: 0.216000
(Iteration 131 / 250) loss: 0.026194
(Epoch 27 / 50) train acc: 1.000000; val_acc: 0.218000
(Epoch 28 / 50) train acc: 1.000000; val_acc: 0.221000
(Iteration 141 / 250) loss: 0.001139
(Epoch 29 / 50) train acc: 1.000000; val_acc: 0.225000
(Epoch 30 / 50) train acc: 1.000000; val_acc: 0.225000
(Iteration 151 / 250) loss: 0.005350
(Epoch 31 / 50) train acc: 1.000000; val_acc: 0.228000
(Epoch 32 / 50) train acc: 1.000000; val_acc: 0.228000
(Iteration 161 / 250) loss: 0.006951
(Epoch 33 / 50) train acc: 1.000000; val_acc: 0.228000
(Epoch 34 / 50) train acc: 1.000000; val_acc: 0.230000
(Iteration 171 / 250) loss: 0.002181
(Epoch 35 / 50) train acc: 1.000000; val_acc: 0.233000
(Epoch 36 / 50) train acc: 1.000000; val_acc: 0.232000
(Iteration 181 / 250) loss: 0.009329
(Epoch 37 / 50) train acc: 1.000000; val_acc: 0.231000
(Epoch 38 / 50) train acc: 1.000000; val_acc: 0.232000
(Iteration 191 / 250) loss: 0.003907
(Epoch 39 / 50) train acc: 1.000000; val_acc: 0.230000
(Epoch 40 / 50) train acc: 1.000000; val_acc: 0.227000
```

```
(Iteration 201 / 250) loss: 0.002780
(Epoch 41 / 50) train acc: 1.000000; val_acc: 0.228000
(Epoch 42 / 50) train acc: 1.000000; val_acc: 0.228000
(Iteration 211 / 250) loss: 0.003743
(Epoch 43 / 50) train acc: 1.000000; val_acc: 0.226000
(Epoch 44 / 50) train acc: 1.000000; val_acc: 0.228000
(Iteration 221 / 250) loss: 0.006557
(Epoch 45 / 50) train acc: 1.000000; val_acc: 0.229000
(Epoch 46 / 50) train acc: 1.000000; val_acc: 0.232000
(Iteration 231 / 250) loss: 0.003235
(Epoch 47 / 50) train acc: 1.000000; val_acc: 0.231000
(Epoch 48 / 50) train acc: 1.000000; val_acc: 0.231000
(Iteration 241 / 250) loss: 0.009860
(Epoch 49 / 50) train acc: 1.000000; val_acc: 0.231000
(Epoch 50 / 50) train acc: 1.000000; val_acc: 0.231000
```

In [16]:
```
# Plotting the loss, training accuracy, and validation accuracy should show clear overf
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()
```



Train a good CNN.

By tweaking different parameters, such as number of convolution layers, learning rate, batch size, etc, you should achieve greater than 62% accuracy on the validation set **with 3 epochs using the sgd_momentum optimizer**.

If you are really careful, you should be able to get nearly 66% accuracy on the validation set. But we don't give extra credits for doing so.

It may take a quite while for your training to be finished. **Do not use more than four convolution layers. Your training shouldn't be longer than one hour.** (This is a rough reference as it depends on the hardware. Our implementation takes about 10-15 minutes to finish.)

Use a large filter/kernel size in the first convolution layer (for example, 7), so you can easily visualize the learend filters.

Since it is relatively slower to train a CNN, you can simply report the best hyper parameters you found. You need report validation accuracy of other choices below.

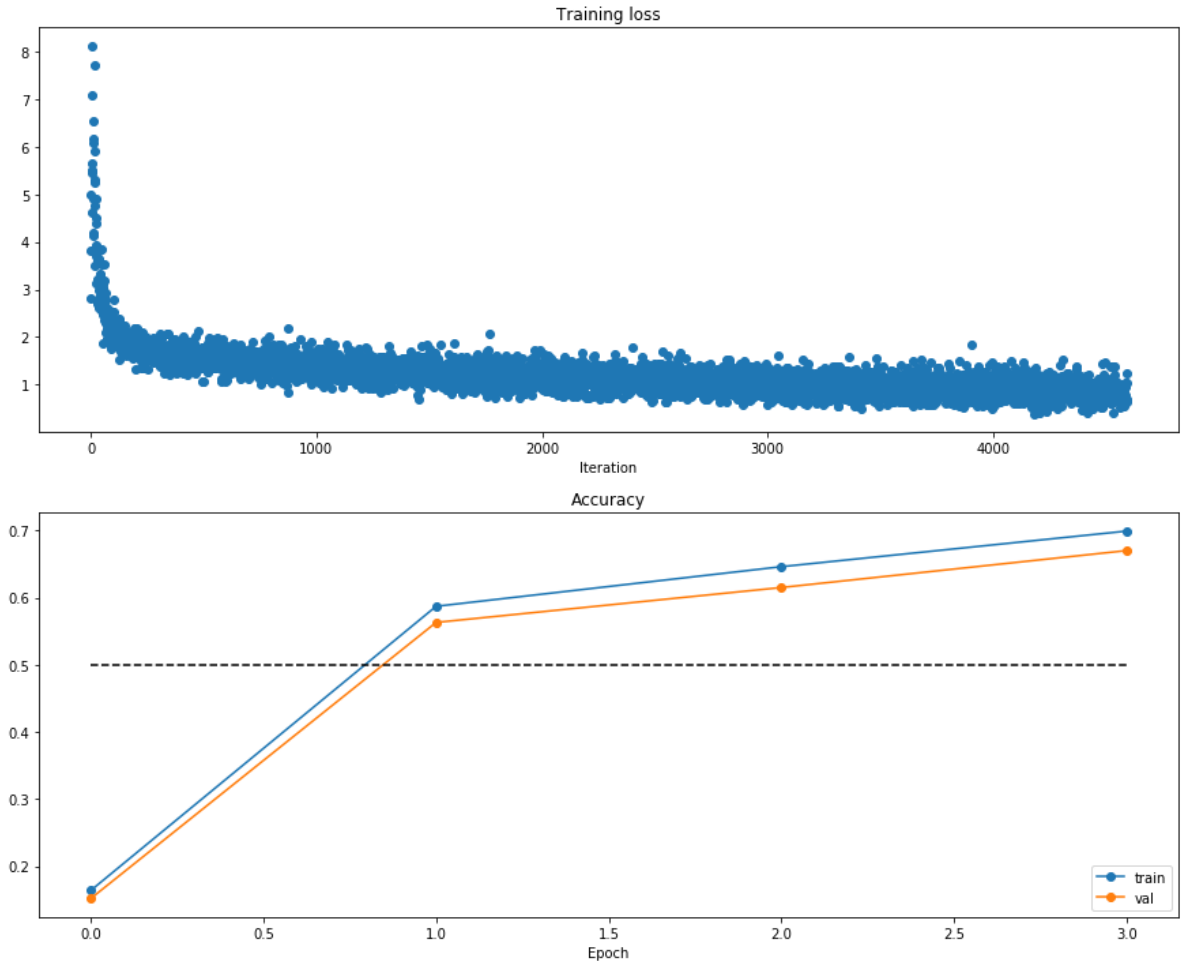
In [17]:
```
best_model = None
################################################################################
# TODO: Train the best ConvNet that you can on CIFAR-10 with 3 epochs using    #
# the sgd_momentum optimizer. Store your best model in the best_model variable.#
################################################################################

# raise NotImplementedError
model = ConvNet(
    weight_scale=0.01,
    reg=0.000001,
    use_batch_norm=True,
    num_filters=[16, 32, 64],
    filter_sizes=[7, 7, 3]
)

solver = Solver(
    model, data,
    num_epochs=3, batch_size=32,
    update_rule='sgd_momentum',
    optim_config={
      'learning_rate': 5e-3, 'lr_decay': 0.98
    },
    verbose=True, print_every=50
)
solver.train()

best_model = model
best_model.params.update(solver.best_params)

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

```
(Iteration 1 / 4593) loss: 2.813290
(Epoch 0 / 3) train acc: 0.164000; val_acc: 0.152000
(Iteration 51 / 4593) loss: 2.896852
(Iteration 101 / 4593) loss: 2.787180
(Iteration 151 / 4593) loss: 1.895524
(Iteration 201 / 4593) loss: 1.753565
(Iteration 251 / 4593) loss: 1.627394
(Iteration 301 / 4593) loss: 1.486855
(Iteration 351 / 4593) loss: 1.639293
(Iteration 401 / 4593) loss: 1.536671
(Iteration 451 / 4593) loss: 1.355593
(Iteration 501 / 4593) loss: 1.456251
(Iteration 551 / 4593) loss: 1.632776
(Iteration 601 / 4593) loss: 1.539111
(Iteration 651 / 4593) loss: 1.255545
(Iteration 701 / 4593) loss: 1.519179
(Iteration 751 / 4593) loss: 1.575602
(Iteration 801 / 4593) loss: 1.313616
(Iteration 851 / 4593) loss: 1.671568
(Iteration 901 / 4593) loss: 1.158067
(Iteration 951 / 4593) loss: 1.283438
(Iteration 1001 / 4593) loss: 1.463887
(Iteration 1051 / 4593) loss: 1.269820
(Iteration 1101 / 4593) loss: 1.189647
(Iteration 1151 / 4593) loss: 1.138343
(Iteration 1201 / 4593) loss: 1.615970
(Iteration 1251 / 4593) loss: 1.102487
(Iteration 1301 / 4593) loss: 1.566165
(Iteration 1351 / 4593) loss: 1.494990
(Iteration 1401 / 4593) loss: 1.460640
(Iteration 1451 / 4593) loss: 1.536115
(Iteration 1501 / 4593) loss: 1.272734
(Epoch 1 / 3) train acc: 0.587000; val_acc: 0.563000
(Iteration 1551 / 4593) loss: 1.398421
(Iteration 1601 / 4593) loss: 1.052416
(Iteration 1651 / 4593) loss: 1.245160
(Iteration 1701 / 4593) loss: 1.186304
(Iteration 1751 / 4593) loss: 1.564447
(Iteration 1801 / 4593) loss: 1.128229
(Iteration 1851 / 4593) loss: 1.132954
(Iteration 1901 / 4593) loss: 0.706591
(Iteration 1951 / 4593) loss: 0.921648
(Iteration 2001 / 4593) loss: 1.324097
(Iteration 2051 / 4593) loss: 1.201326
(Iteration 2101 / 4593) loss: 1.227291
(Iteration 2151 / 4593) loss: 0.879298
(Iteration 2201 / 4593) loss: 0.779294
(Iteration 2251 / 4593) loss: 1.420853
(Iteration 2301 / 4593) loss: 1.291180
(Iteration 2351 / 4593) loss: 0.956255
(Iteration 2401 / 4593) loss: 0.884758
(Iteration 2451 / 4593) loss: 1.231592
(Iteration 2501 / 4593) loss: 1.024643
(Iteration 2551 / 4593) loss: 0.646960
(Iteration 2601 / 4593) loss: 1.542949
(Iteration 2651 / 4593) loss: 1.226053
(Iteration 2701 / 4593) loss: 1.144972
(Iteration 2751 / 4593) loss: 0.916160
(Iteration 2801 / 4593) loss: 1.375703
(Iteration 2851 / 4593) loss: 1.016102
(Iteration 2901 / 4593) loss: 0.801092
```

```
(Iteration 2951 / 4593) loss: 0.950697
(Iteration 3001 / 4593) loss: 0.851425
(Iteration 3051 / 4593) loss: 1.087421
(Epoch 2 / 3) train acc: 0.646000; val_acc: 0.615000
(Iteration 3101 / 4593) loss: 0.762459
(Iteration 3151 / 4593) loss: 0.904069
(Iteration 3201 / 4593) loss: 0.953213
(Iteration 3251 / 4593) loss: 1.128821
(Iteration 3301 / 4593) loss: 1.024762
(Iteration 3351 / 4593) loss: 0.863470
(Iteration 3401 / 4593) loss: 1.193716
(Iteration 3451 / 4593) loss: 1.085220
(Iteration 3501 / 4593) loss: 0.973071
(Iteration 3551 / 4593) loss: 1.101911
(Iteration 3601 / 4593) loss: 0.873897
(Iteration 3651 / 4593) loss: 1.025374
(Iteration 3701 / 4593) loss: 0.916968
(Iteration 3751 / 4593) loss: 0.921548
(Iteration 3801 / 4593) loss: 0.742956
(Iteration 3851 / 4593) loss: 0.544873
(Iteration 3901 / 4593) loss: 0.781828
(Iteration 3951 / 4593) loss: 0.543661
(Iteration 4001 / 4593) loss: 0.904803
(Iteration 4051 / 4593) loss: 1.390126
(Iteration 4101 / 4593) loss: 0.853765
(Iteration 4151 / 4593) loss: 1.148712
(Iteration 4201 / 4593) loss: 0.996141
(Iteration 4251 / 4593) loss: 0.835008
(Iteration 4301 / 4593) loss: 0.537155
(Iteration 4351 / 4593) loss: 1.131438
(Iteration 4401 / 4593) loss: 0.696243
(Iteration 4451 / 4593) loss: 0.706256
(Iteration 4501 / 4593) loss: 0.685065
(Iteration 4551 / 4593) loss: 0.815492
(Epoch 3 / 3) train acc: 0.699000; val_acc: 0.670000
```

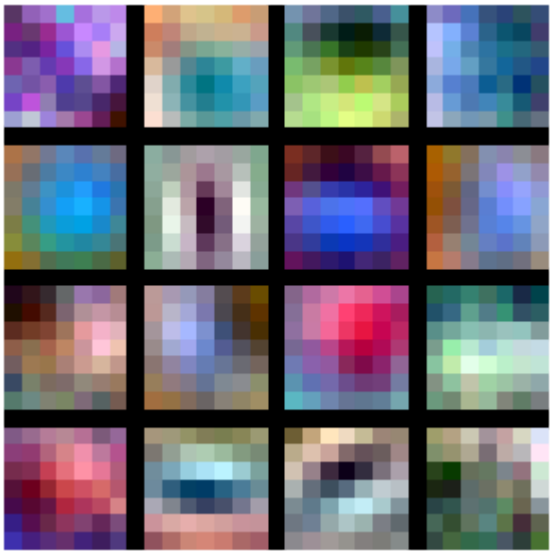
In [18]:
```
# Run your best model on the validation and test sets. You should achieve above 62% acc
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())


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()
```

```
Validation set accuracy:  0.672
Test set accuracy:  0.666
```



Visualize Filters

In [19]:
```
# You can visualize the first-layer convolutional filters from the trained network by r
from cs6353.vis_utils import visualize_grid

grid = visualize_grid(model.params['W_1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



Report validation accuray for other hyper parameters you have tried

1. weight_scale=0.01, reg=0.000001,num_filters= [32, 32, 64], filter_sizes= [7, 7, 3], learning_rate = 5e-2, validation accuray = 0.18
2. weight_scale=0.01, reg=0.00001,num_filters= [16, 32, 64],filter_sizes= [7, 3, 3], learning_rate = 5e-2, validation accuray = 0.13

Task 4.4: train a ConvNet without using batch normalization layers.

Report the best validation accuracy you can get and discuss how it is different from the version with batch normalization layers.

1. With batch normalization:0.672
2. Without batch normalization: 0.61
3. Different: The batch normalization version has more accurency than without batch normalization version. This is because it is convergence faster than no batch normalization version.

In [ ]:
```
```