Optimization for Fully Connected Networks

In this notebook, we will implement different optimization rules for gradient descent. We have provided starter code; however, you will need to copy and paste your code from your implementation of the modular fully connected nets in HW #3 to build upon this.

If you did not complete `affine` forward and backwards passes, or `relu` forward and backward passes from HW #3 correctly, you may use another classmate's implementation of these functions for this assignment, or contact us at ece239as.w18@gmail.com.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [2]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numer
ical_gradient_array
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-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 in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

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

Building upon your HW #3 implementation

Copy and paste the following functions from your HW #3 implementation of a modular FC net:

- `affine_forward` in `nndl/layers.py`
- `affine_backward` in `nndl/layers.py`
- `relu_forward` in `nndl/layers.py`
- `relu_backward` in `nndl/layers.py`
- `affine_relu_forward` in `nndl/layer_utils.py`
- `affine_relu_backward` in `nndl/layer_utils.py`
- The `FullyConnectedNet` class in `nndl/fc_net.py`

Test all functions you copy and pasted

In [5]:
```
from nndl.layer_tests import *

affine_forward_test(); print('\n')
affine_backward_test(); print('\n')
relu_forward_test(); print('\n')
relu_backward_test(); print('\n')
affine_relu_test(); print('\n')
fc_net_test()
```

```
If affine_forward function is working, difference should be less than
1e-9:
difference: 9.76984946819e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 4.67448335325e-10
dw error: 4.43584616548e-10
db error: 4.44843750402e-11


If relu_forward function is working, difference should be around 1e-
8:
difference: 4.99999979802e-08


If relu_forward function is working, error should be less than 1e-9:
dx error: 1.89289326869e-11


If affine_relu_forward and affine_relu_backward are working, error sh
ould be less than 1e-9::
dx error: 5.25709056874e-10
dw error: 2.76768228346e-09
db error: 7.82663359068e-12


Running check with reg = 0
Initial loss: 2.30500113891
W1 relative error: 1.29738909385e-05
W2 relative error: 3.90814600267e-05
W3 relative error: 2.91561284105e-07
b1 relative error: 1.01500264396e-08
b2 relative error: 3.08118185084e-09
b3 relative error: 9.56576884895e-11
Running check with reg = 3.14
Initial loss: 6.77006515357
W1 relative error: 1.90337580641e-07
W2 relative error: 1.11940268416e-08
W3 relative error: 6.1487567534e-09
b1 relative error: 1.62225809703e-08
b2 relative error: 5.94630940736e-09
b3 relative error: 2.56119146973e-10
```

Training a larger model

In general, proceeding with vanilla stochastic gradient descent to optimize models may be fraught with problems and limitations, as discussed in class. Thus, we implement optimizers that improve on SGD.

SGD + momentum

In the following section, implement SGD with momentum. Read the `nndl/optim.py` API, which is provided by CS231n, and be sure you understand it. After, implement `sgd_momentum` in `nndl/optim.py`. Test your implementation of `sgd_momentum` by running the cell below.

In [8]:
```
from nndl.optim import sgd_momentum

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

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

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

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

```
next_w error: 8.88234703351e-09
velocity error: 4.26928774328e-09
```

SGD + Nesterov momentum

Implement `sgd_nesterov_momentum` in `ndl/optim.py`.

In [10]:
```
from nndl.optim import sgd_nesterov_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_nesterov_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [0.08714,      0.15246105,  0.21778211,  0.28310316,  0.34842421],
  [0.41374526,   0.47906632,  0.54438737,  0.60970842,  0.67502947],
  [0.74035053,   0.80567158,  0.87099263,  0.93631368,  1.00163474],
  [1.06695579,   1.13227684,  1.19759789,  1.26291895,  1.32824   ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579,  0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

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

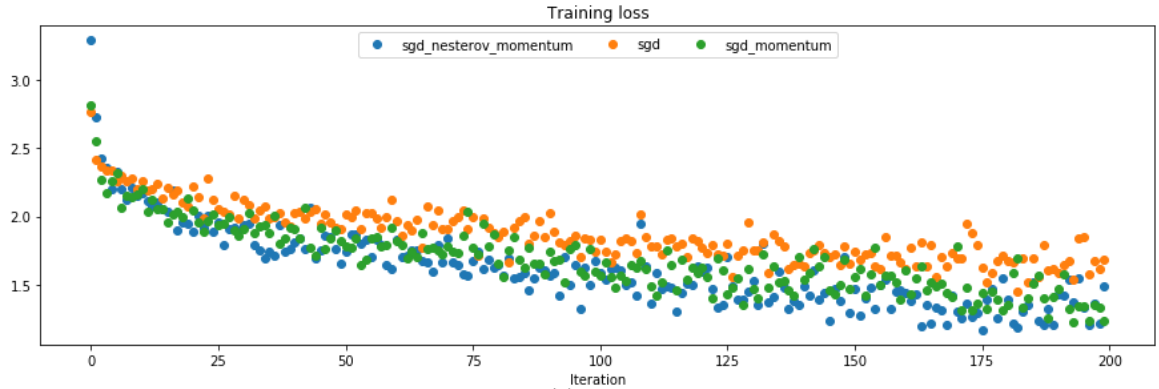
```
next_w error: 1.08751868451e-08
velocity error: 4.26928774328e-09
```

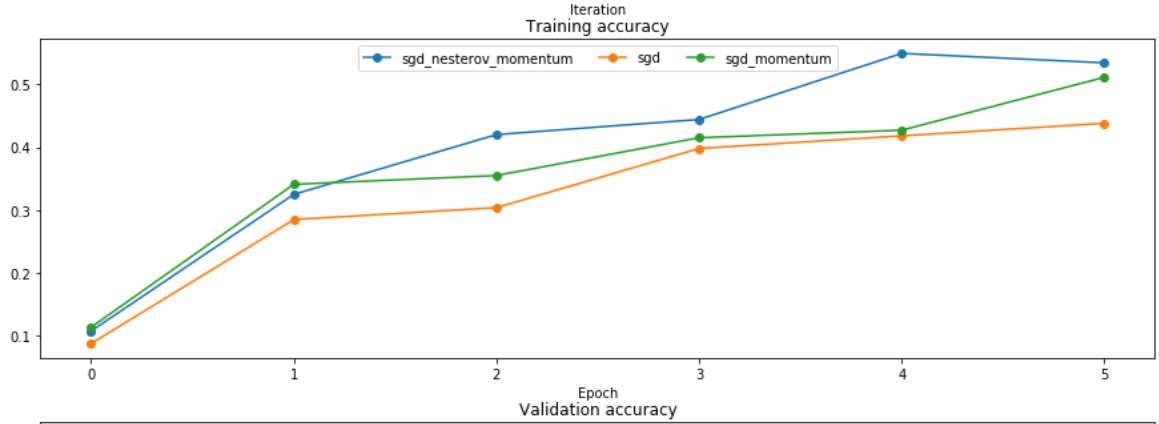
Evaluating SGD, SGD+Momentum, and SGD+NesterovMomentum

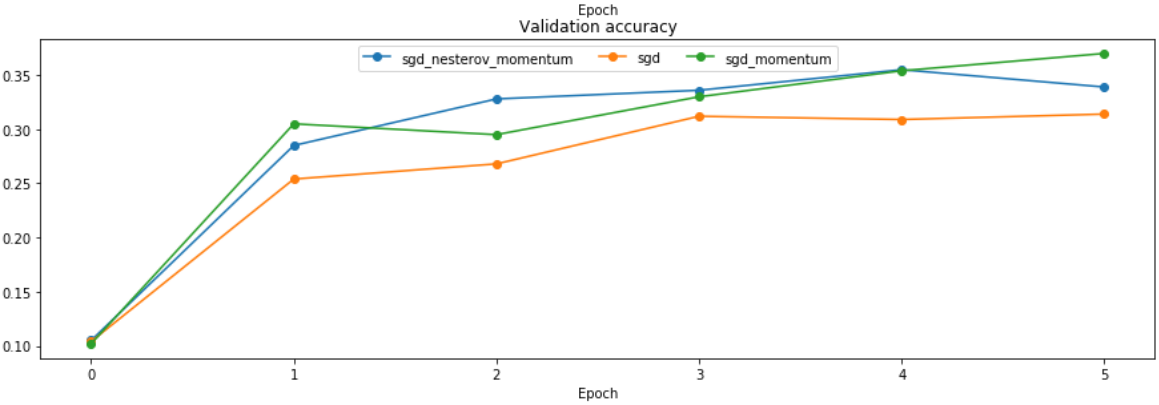
Run the following cell to train a 6 layer FC net with SGD, SGD+momentum, and SGD+Nesterov momentum. You should see that SGD+momentum achieves a better loss than SGD, and that SGD+Nesterov momentum achieves a slightly better loss (and training accuracy) than SGD+momentum.

In [11]:
```
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', 'sgd_nesterov_momentum']:
  print('Optimizing with {}'.format(update_rule))
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5
e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=False)
  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 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()
```

```
Optimizing with sgd

Optimizing with sgd_momentum

Optimizing with sgd_nesterov_momentum


/home/ben/Documents/239AS/HW4/local/lib/python2.7/site-packages/matpl
otlib/cbook/deprecation.py:106: MatplotlibDeprecationWarning: Adding
 an axes using the same arguments as a previous axes currently reuses
the earlier instance.  In a future version, a new instance will alway
s be created and returned.  Meanwhile, this warning can be suppresse
d, and the future behavior ensured, by passing a unique label to each
axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```







RMSProp

Now we go to techniques that adapt the gradient. Implement `rmsprop` in `nndl/optim.py`. Test your implementation by running the cell below.

In [12]:
```
from nndl.optim import rmsprop

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

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

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

print('next_w error: {}'.format(rel_error(expected_next_w, next_w)))
print('cache error: {}'.format(rel_error(expected_cache,
config['a'])))
```

```
next_w error: 9.52468751104e-08
cache error: 2.64779558072e-09
```

Adaptive moments

Now, implement `adam` in `nndl/optim.py`. Test your implementation by running the cell below.

In [20]:
```
# Test Adam implementation; you should see errors around 1e-7 or less
from nndl.optim import adam

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

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

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

print('next_w error: {}'.format(rel_error(expected_next_w, next_w)))
print('a error: {}'.format(rel_error(expected_a, config['a'])))
print('v error: {}'.format(rel_error(expected_v, config['v'])))
```

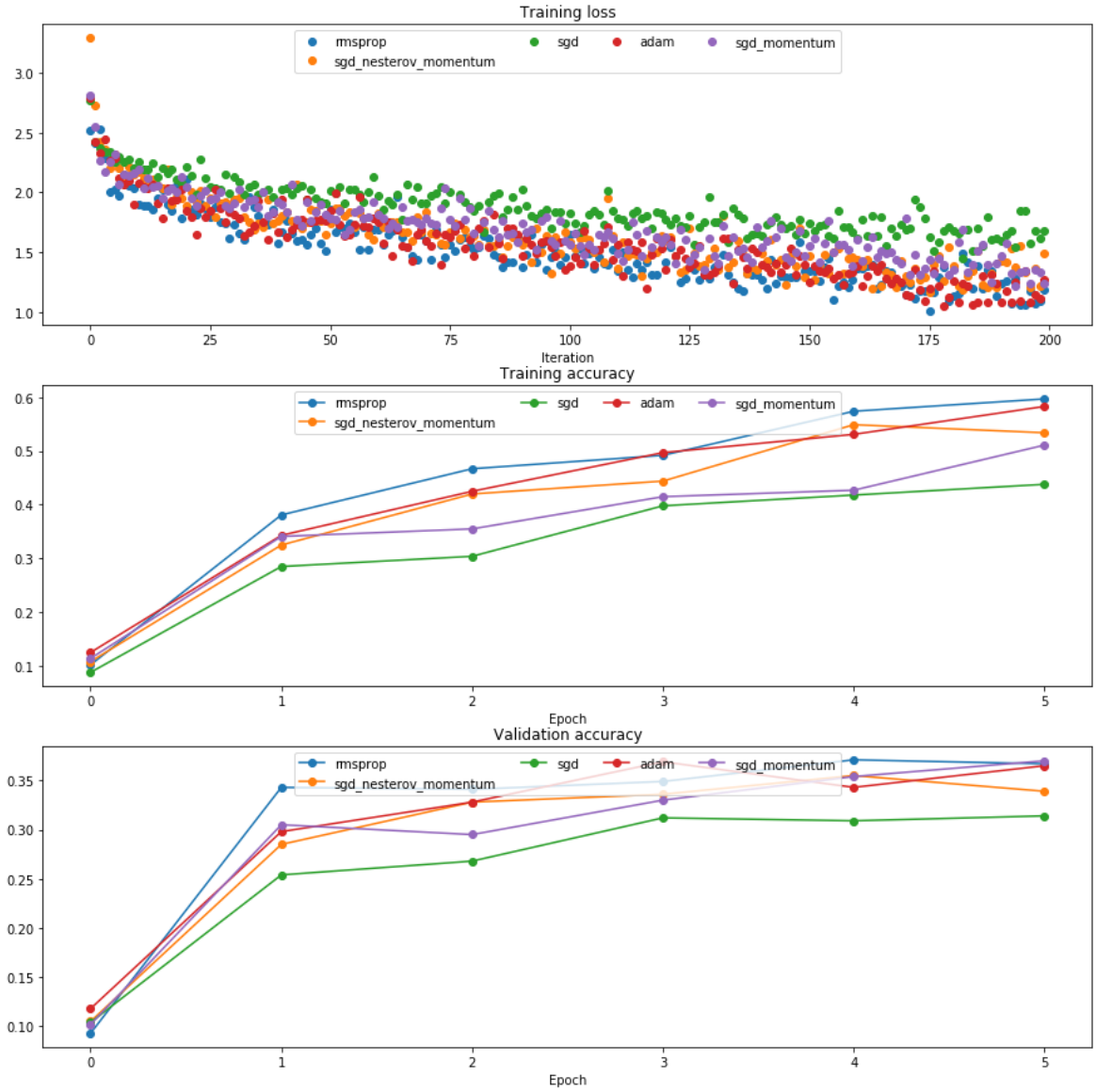
```
next_w error: 1.13956917985e-07
a error: 4.20831403811e-09
v error: 4.21496319311e-09
```

Comparing SGD, SGD+NesterovMomentum, RMSProp, and Adam

The following code will compare optimization with SGD, Momentum, Nesterov Momentum, RMSProp and Adam. In our code, we find that RMSProp, Adam, and SGD + Nesterov Momentum achieve approximately the same training error after a few training epochs.

In [22]:
```
learning_rates = {'rmsprop': 2e-4, 'adam': 1e-3}

for update_rule in ['adam', 'rmsprop']:
  print('Optimizing with {}'.format(update_rule))
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5
e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=False)
  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 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()
```

```
Optimizing with adam

Optimizing with rmsprop
```



Easier optimization

In the following cell, we'll train a 4 layer neural network having 500 units in each hidden layer with the different optimizers, and find that it is far easier to get up to 50+% performance on CIFAR-10. After we implement batchnorm and dropout, we'll ask you to get 60+% on CIFAR-10.

In [23]:
```
optimizer = 'adam'
best_model = None

layer_dims = [500, 500, 500]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale,
                          use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=10, batch_size=100,
                update_rule=optimizer,
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=lr_decay,
                verbose=True, print_every=50)
solver.train()
```

```
(Iteration 1 / 4900) loss: 2.298090
(Epoch 0 / 10) train acc: 0.169000; val_acc: 0.162000
(Iteration 51 / 4900) loss: 1.781161
(Iteration 101 / 4900) loss: 1.728707
(Iteration 151 / 4900) loss: 1.858667
(Iteration 201 / 4900) loss: 1.644391
(Iteration 251 / 4900) loss: 1.615825
(Iteration 301 / 4900) loss: 1.622498
(Iteration 351 / 4900) loss: 1.747485
(Iteration 401 / 4900) loss: 1.552316
(Iteration 451 / 4900) loss: 1.768418
(Epoch 1 / 10) train acc: 0.460000; val_acc: 0.425000
(Iteration 501 / 4900) loss: 1.742538
(Iteration 551 / 4900) loss: 1.581496
(Iteration 601 / 4900) loss: 1.333104
(Iteration 651 / 4900) loss: 1.337607
(Iteration 701 / 4900) loss: 1.567914
(Iteration 751 / 4900) loss: 1.639876
(Iteration 801 / 4900) loss: 1.555322
(Iteration 851 / 4900) loss: 1.448859
(Iteration 901 / 4900) loss: 1.581548
(Iteration 951 / 4900) loss: 1.391528
(Epoch 2 / 10) train acc: 0.524000; val_acc: 0.468000
(Iteration 1001 / 4900) loss: 1.411459
(Iteration 1051 / 4900) loss: 1.434459
(Iteration 1101 / 4900) loss: 1.468449
(Iteration 1151 / 4900) loss: 1.439805
(Iteration 1201 / 4900) loss: 1.328831
(Iteration 1251 / 4900) loss: 1.601632
(Iteration 1301 / 4900) loss: 1.293863
(Iteration 1351 / 4900) loss: 1.434035
(Iteration 1401 / 4900) loss: 1.396651
(Iteration 1451 / 4900) loss: 1.483837
(Epoch 3 / 10) train acc: 0.527000; val_acc: 0.484000
(Iteration 1501 / 4900) loss: 1.277409
(Iteration 1551 / 4900) loss: 1.435540
(Iteration 1601 / 4900) loss: 1.288683
(Iteration 1651 / 4900) loss: 1.296424
(Iteration 1701 / 4900) loss: 1.278664
(Iteration 1751 / 4900) loss: 1.308277
(Iteration 1801 / 4900) loss: 1.389170
(Iteration 1851 / 4900) loss: 1.257221
(Iteration 1901 / 4900) loss: 1.284852
(Iteration 1951 / 4900) loss: 1.405904
(Epoch 4 / 10) train acc: 0.554000; val_acc: 0.514000
(Iteration 2001 / 4900) loss: 1.181688
(Iteration 2051 / 4900) loss: 1.200336
(Iteration 2101 / 4900) loss: 1.240947
(Iteration 2151 / 4900) loss: 1.225776
(Iteration 2201 / 4900) loss: 1.353713
(Iteration 2251 / 4900) loss: 1.078632
(Iteration 2301 / 4900) loss: 1.307681
(Iteration 2351 / 4900) loss: 1.395831
(Iteration 2401 / 4900) loss: 1.297470
(Epoch 5 / 10) train acc: 0.557000; val_acc: 0.516000
(Iteration 2451 / 4900) loss: 1.135156
(Iteration 2501 / 4900) loss: 1.457520
```

```
(Iteration 2551 / 4900) loss: 1.051139
(Iteration 2601 / 4900) loss: 1.221754
(Iteration 2651 / 4900) loss: 1.078154
(Iteration 2701 / 4900) loss: 1.101139
(Iteration 2751 / 4900) loss: 1.247272
(Iteration 2801 / 4900) loss: 1.034844
(Iteration 2851 / 4900) loss: 1.284162
(Iteration 2901 / 4900) loss: 1.167326
(Epoch 6 / 10) train acc: 0.616000; val_acc: 0.530000
(Iteration 2951 / 4900) loss: 1.076583
(Iteration 3001 / 4900) loss: 0.944791
(Iteration 3051 / 4900) loss: 0.989512
(Iteration 3101 / 4900) loss: 0.919589
(Iteration 3151 / 4900) loss: 1.250447
(Iteration 3201 / 4900) loss: 0.948299
(Iteration 3251 / 4900) loss: 1.125616
(Iteration 3301 / 4900) loss: 1.075672
(Iteration 3351 / 4900) loss: 1.039359
(Iteration 3401 / 4900) loss: 1.032512
(Epoch 7 / 10) train acc: 0.583000; val_acc: 0.520000
(Iteration 3451 / 4900) loss: 1.055815
(Iteration 3501 / 4900) loss: 1.155710
(Iteration 3551 / 4900) loss: 1.487554
(Iteration 3601 / 4900) loss: 1.087278
(Iteration 3651 / 4900) loss: 1.028502
(Iteration 3701 / 4900) loss: 1.170369
(Iteration 3751 / 4900) loss: 0.920450
(Iteration 3801 / 4900) loss: 0.978088
(Iteration 3851 / 4900) loss: 0.915175
(Iteration 3901 / 4900) loss: 1.050658
(Epoch 8 / 10) train acc: 0.626000; val_acc: 0.534000
(Iteration 3951 / 4900) loss: 1.089473
(Iteration 4001 / 4900) loss: 1.234826
(Iteration 4051 / 4900) loss: 1.025201
(Iteration 4101 / 4900) loss: 0.867859
(Iteration 4151 / 4900) loss: 1.154162
(Iteration 4201 / 4900) loss: 0.922180
(Iteration 4251 / 4900) loss: 0.850985
(Iteration 4301 / 4900) loss: 1.123788
(Iteration 4351 / 4900) loss: 1.068928
(Iteration 4401 / 4900) loss: 0.848677
(Epoch 9 / 10) train acc: 0.668000; val_acc: 0.537000
(Iteration 4451 / 4900) loss: 0.850790
(Iteration 4501 / 4900) loss: 0.914570
(Iteration 4551 / 4900) loss: 0.775186
(Iteration 4601 / 4900) loss: 0.835760
(Iteration 4651 / 4900) loss: 0.806960
(Iteration 4701 / 4900) loss: 1.004544
(Iteration 4751 / 4900) loss: 1.019356
(Iteration 4801 / 4900) loss: 0.766119
(Iteration 4851 / 4900) loss: 0.803193
(Epoch 10 / 10) train acc: 0.680000; val_acc: 0.538000
```

In [24]:
```
y_test_pred = np.argmax(model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(model.loss(data['X_val']), axis=1)
print('Validation set accuracy: {}'.format(np.mean(y_val_pred == data['y_val'])))
print('Test set accuracy: {}'.format(np.mean(y_test_pred == data['y_test'])))
```

```
Validation set accuracy: 0.538
Test set accuracy: 0.553
```

```python
import numpy as np

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

"""
This file implements various first-order update rules that are commonly used for
training neural networks. Each update rule accepts current weights and the
gradient of the loss with respect to those weights and produces the next set of
weights. Each update rule has the same interface:

def update(w, dw, config=None):

Inputs:
  - w: A numpy array giving the current weights.
  - dw: A numpy array of the same shape as w giving the gradient of the
    loss with respect to w.
  - config: A dictionary containing hyperparameter values such as learning rate,
    momentum, etc. If the update rule requires caching values over many
    iterations, then config will also hold these cached values.

Returns:
  - next_w: The next point after the update.
  - config: The config dictionary to be passed to the next iteration of the
    update rule.

NOTE: For most update rules, the default learning rate will probably not perform
well; however the default values of the other hyperparameters should work well
for a variety of different problems.

For efficiency, update rules may perform in-place updates, mutating w and
setting next_w equal to w.
"""


def sgd(w, dw, config=None):
  """
  Performs vanilla stochastic gradient descent.

  config format:
  - learning_rate: Scalar learning rate.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)

  w -= config['learning_rate'] * dw
  return w, config


def sgd_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
```

```
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and store the updated velocity as v.
  # ================================================================ #

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

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  config['velocity'] = v

  return next_w, config

def sgd_nesterov_momentum(w, dw, config=None):
  """
  Performs stochastic gradient descent with Nesterov momentum.

  config format:
  - learning_rate: Scalar learning rate.
  - momentum: Scalar between 0 and 1 giving the momentum value.
    Setting momentum = 0 reduces to sgd.
  - velocity: A numpy array of the same shape as w and dw used to store a moving
    average of the gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
  v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the momentum update formula.  Return the updated weights
  #   as next_w, and store the updated velocity as v.
  # ================================================================ #

  v_new = config['momentum']*v - config['learning_rate']*dw
  next_w = w + v_new + config['momentum']*(v_new - v)
  v = v_new

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  config['velocity'] = v

  return next_w, config

def rmsprop(w, dw, config=None):
  """
  Uses the RMSProp update rule, which uses a moving average of squared gradient
  values to set adaptive per-parameter learning rates.

  config format:
  - learning_rate: Scalar learning rate.
  - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
    gradient cache.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - beta: Moving average of second moments of gradients.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-2)
  config.setdefault('decay_rate', 0.99)
```

```
  config.setdefault('epsilon', 1e-8)
  config.setdefault('a', np.zeros_like(w))

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement RMSProp.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, so they can be used for future gradients. Concretely,
  #   config['a'] corresponds to "a" in the lecture notes.
  # ================================================================ #
  dr = config['decay_rate']
  a = config['a']
  lr = config['learning_rate']
  eps = config['epsilon']

  a = dr*a + (1 - dr) * np.square(dw)

  next_w = w - lr*np.multiply(1/(np.sqrt(a)+eps),dw)


  config['a'] = a


  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return next_w, config


def adam(w, dw, config=None):
  """
  Uses the Adam update rule, which incorporates moving averages of both the
  gradient and its square and a bias correction term.

  config format:
  - learning_rate: Scalar learning rate.
  - beta1: Decay rate for moving average of first moment of gradient.
  - beta2: Decay rate for moving average of second moment of gradient.
  - epsilon: Small scalar used for smoothing to avoid dividing by zero.
  - m: Moving average of gradient.
  - v: Moving average of squared gradient.
  - t: Iteration number.
  """
  if config is None: config = {}
  config.setdefault('learning_rate', 1e-3)
  config.setdefault('beta1', 0.9)
  config.setdefault('beta2', 0.999)
  config.setdefault('epsilon', 1e-8)
  config.setdefault('v', np.zeros_like(w))
  config.setdefault('a', np.zeros_like(w))
  config.setdefault('t', 0)

  next_w = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement Adam.  Store the next value of w as next_w.  You need
  #   to also store in config['a'] the moving average of the second
  #   moment gradients, and in config['v'] the moving average of the
  #   first moments.  Finally, store in config['t'] the increasing time.
  # ================================================================ #
  lr = config['learning_rate']
  b1 = config['beta1']
  b2 = config['beta2']
  eps = config['epsilon']
  v = config['v']
```

```
208   a = config['a']
209   t = config['t']
210   t = t+1
211
212   v = b1*v + (1-b1)*dw
213   a = b2*a + (1-b2)*np.square(dw)
214
215   v_mod = 1/(1-np.power(b1, t))*v
216   a_mod = 1/(1-np.power(b2, t))*a
217
218   next_w = w - lr*np.multiply(1/(np.sqrt(a_mod)+eps),v_mod)
219
220   config['a'] = a
221   config['v'] = v
222   config['t'] = t
223   # ============================================================ #
224   # END YOUR CODE HERE
225   # ============================================================ #
226
227   return next_w, config
228
229
230
231
```

Batch Normalization

In this notebook, you will implement the batch normalization layers of a neural network to increase its performance. If you have any confusion, please review the details of batch normalization from the lecture notes.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from nndl.layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numer
ical_gradient_array
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-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 [2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

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

Batchnorm forward pass

Implement the training time batchnorm forward pass, `batchnorm_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

In [3]:
```
# Check the training-time forward pass by checking means and variance
s
# of features both before and after batch normalization

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

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

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

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

```
Before batch normalization:
('  means: ', array([-41.70415321, -32.10771629,   7.67615128]))
('  stds: ', array([36.80109114, 34.3529545 , 31.79014897]))
After batch normalization (gamma=1, beta=0)
('  mean: ', array([-1.84297022e-16,  4.40758541e-16, -8.88178420e-1
7]))
('  std: ', array([1., 1., 1.]))
After batch normalization (nontrivial gamma, beta)
('  means: ', array([11., 12., 13.]))
('  stds: ', array([1.        , 1.99999999, 2.99999999]))
```

Implement the testing time batchnorm forward pass, `batchnorm_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

In [4]:
```
# 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, 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 np.arange(50):
  X = np.random.randn(N, D1)
  a = np.maximum(0, X.dot(W1)).dot(W2)
  batchnorm_forward(a, gamma, beta, bn_param)
bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

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

```
After batch normalization (test-time):
('  means: ', array([ 0.11621084,  0.02637013, -0.00563659]))
('  stds: ', array([0.93173868, 0.99679166, 0.97778236]))
```

Batchnorm backward pass

Implement the backward pass for the batchnorm layer, `batchnorm_backward` in `nndl/layers.py`. Check your implementation by running the following cell.

In [5]:
```
# Gradient check batchnorm backward pass

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, gamma, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, beta, bn_param)[0]

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

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
('dx error: ', 6.307803224861234e-10)
('dgamma error: ', 6.733907866852426e-11)
('dbeta error: ', 3.2755938359331305e-12)
```

Implement a fully connected neural network with batchnorm layers

Modify the `FullyConnectedNet()` class in `nndl/fc_net.py` to incorporate batchnorm layers. You will need to modify the class in the following areas:

(1) The gammas and betas need to be initialized to 1's and 0's respectively in `__init__`.

(2) The `batchnorm_forward` layer needs to be inserted between each affine and relu layer (except in the output layer) in a forward pass computation in `loss`. You may find it helpful to write an `affine_batchnorm_relu()` layer in `nndl/layer_utils.py` although this is not necessary.

(3) The `batchnorm_backward` layer has to be appropriately inserted when calculating gradients.

After you have done the appropriate modifications, check your implementation by running the following cell.

Note, while the relative error for W3 should be small, as we backprop gradients more, you may find the relative error increases. Our relative error for W1 is on the order of 1e-4.

In [8]:
```
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.floa
t64,
                            use_batchnorm=True)

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

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name],
verbose=False, h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num, gr
ads[name])))
  if reg == 0: print('\n')
```

```
('Running check with reg = ', 0)
('Initial loss: ', 2.2283464235970376)
W1 relative error: 7.13349664925e-05
W2 relative error: 5.88482583979e-06
W3 relative error: 4.13841203778e-10
b1 relative error: 5.55111512313e-09
b2 relative error: 2.22044604925e-08
b3 relative error: 7.63662765824e-11
beta1 relative error: 5.77128481001e-09
beta2 relative error: 1.4019912764e-09
gamma1 relative error: 6.26605330605e-09
gamma2 relative error: 3.81841860049e-09


('Running check with reg = ', 3.14)
('Initial loss: ', 7.198055088577592)
W1 relative error: 0.000421468416917
W2 relative error: 2.27190091105e-06
W3 relative error: 2.07679342463e-07
b1 relative error: 1.11022302463e-07
b2 relative error: 1.99840144433e-07
b3 relative error: 1.15102423709e-10
beta1 relative error: 1.81144740385e-08
beta2 relative error: 3.8304116936e-08
gamma1 relative error: 3.31514534267e-08
gamma2 relative error: 9.75143468982e-09
```

Training a deep fully connected network with batch normalization.

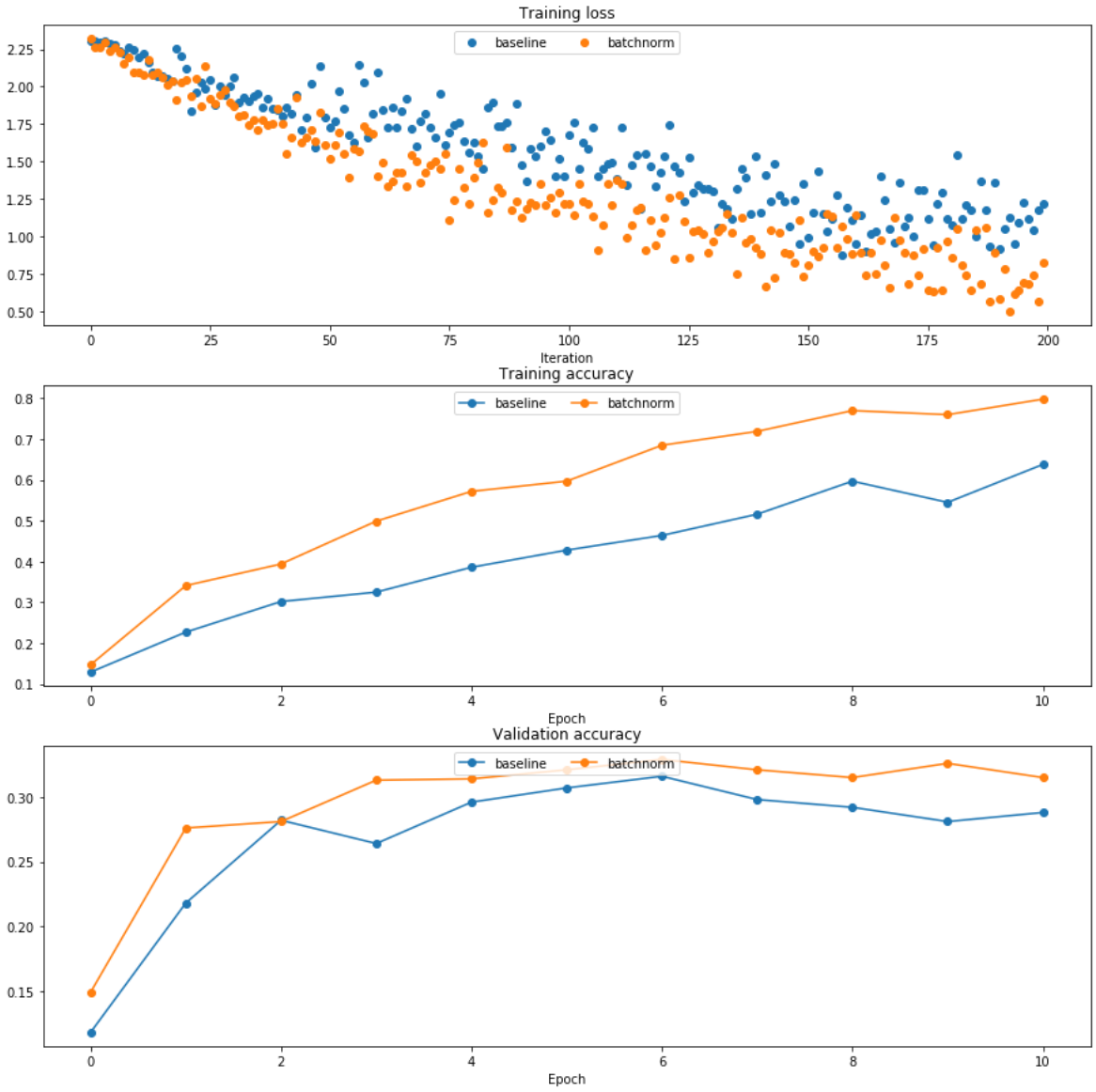
To see if batchnorm helps, let's train a deep neural network with and without batch normalization.

In [9]:
```
# Try training a very deep net with batchnorm
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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

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

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

```
(Iteration 1 / 200) loss: 2.318586
(Epoch 0 / 10) train acc: 0.147000; val_acc: 0.149000
(Epoch 1 / 10) train acc: 0.341000; val_acc: 0.276000
(Epoch 2 / 10) train acc: 0.394000; val_acc: 0.281000
(Epoch 3 / 10) train acc: 0.499000; val_acc: 0.313000
(Epoch 4 / 10) train acc: 0.572000; val_acc: 0.314000
(Epoch 5 / 10) train acc: 0.597000; val_acc: 0.321000
(Epoch 6 / 10) train acc: 0.685000; val_acc: 0.329000
(Epoch 7 / 10) train acc: 0.719000; val_acc: 0.321000
(Epoch 8 / 10) train acc: 0.770000; val_acc: 0.315000
(Epoch 9 / 10) train acc: 0.760000; val_acc: 0.326000
(Epoch 10 / 10) train acc: 0.798000; val_acc: 0.315000
(Iteration 1 / 200) loss: 2.302658
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.118000
(Epoch 1 / 10) train acc: 0.227000; val_acc: 0.218000
(Epoch 2 / 10) train acc: 0.302000; val_acc: 0.282000
(Epoch 3 / 10) train acc: 0.325000; val_acc: 0.264000
(Epoch 4 / 10) train acc: 0.386000; val_acc: 0.296000
(Epoch 5 / 10) train acc: 0.428000; val_acc: 0.307000
(Epoch 6 / 10) train acc: 0.464000; val_acc: 0.316000
(Epoch 7 / 10) train acc: 0.516000; val_acc: 0.298000
(Epoch 8 / 10) train acc: 0.597000; val_acc: 0.292000
(Epoch 9 / 10) train acc: 0.545000; val_acc: 0.281000
(Epoch 10 / 10) train acc: 0.638000; val_acc: 0.288000
```

In [10]:
```
plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

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

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

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

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

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

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

```
/home/ben/Documents/239AS/HW4/local/lib/python2.7/site-packages/matpl
otlib/cbook/deprecation.py:106: MatplotlibDeprecationWarning: Adding
 an axes using the same arguments as a previous axes currently reuses
the earlier instance.  In a future version, a new instance will alway
s be created and returned.  Meanwhile, this warning can be suppresse
d, and the future behavior ensured, by passing a unique label to each
axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



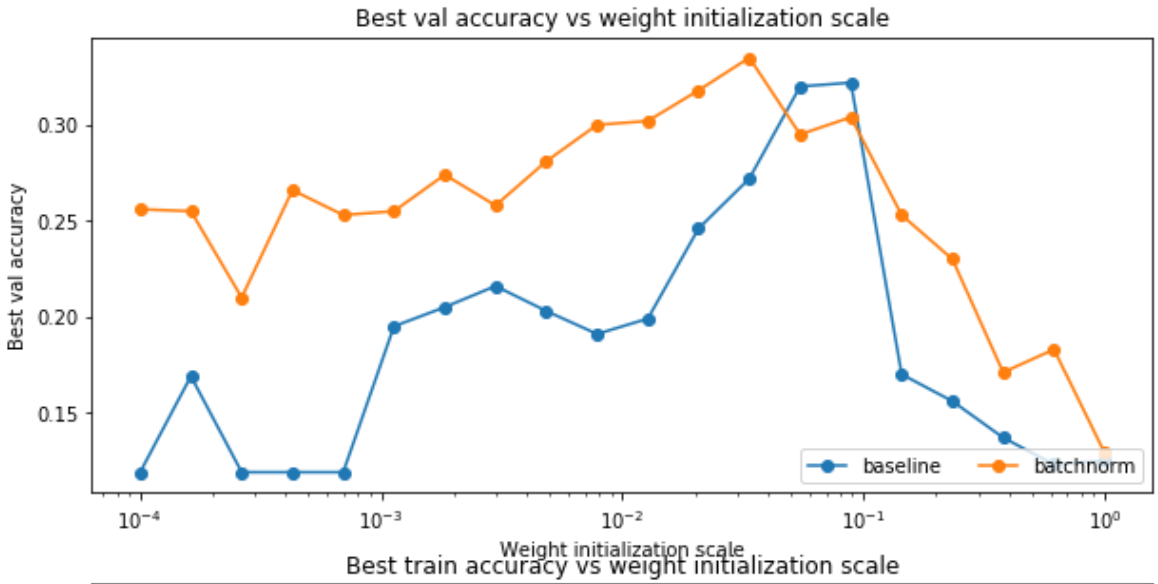
Batchnorm and initialization

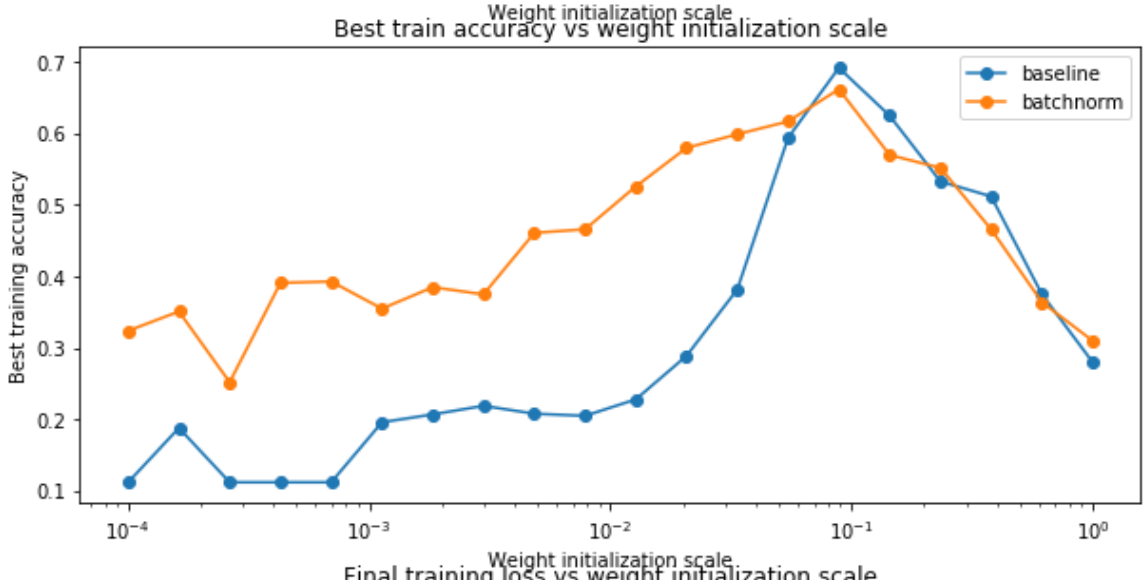
The following cells run an experiment where for a deep network, the initialization is varied. We do training for when batchnorm layers are and are not included.

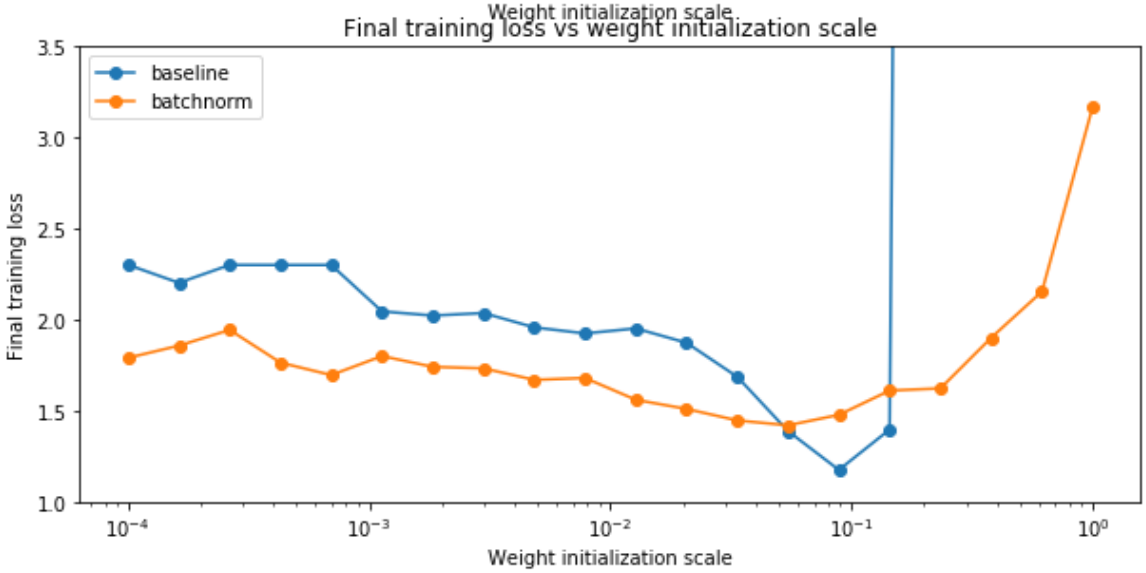
In [13]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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 = {}
solvers = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale {} / {}'.format(i + 1, len(weight_scale
s)))
  bn_model = FullyConnectedNet(hidden_dims,
weight_scale=weight_scale, use_batchnorm=True)
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, u
se_batchnorm=False)

  bn_solver = Solver(bn_model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
  bn_solver.train()
  bn_solvers[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[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 [23]:
```
# 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].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers[ws].train_acc_history))

  best_val_accs.append(max(solvers[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers[ws].loss_history[-100:]))

bn_final_train_loss.append(np.mean(bn_solvers[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='batchnor
m')
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='batchno
rm')
plt.legend()
plt.gca().set_ylim(1.0, 3.5)

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







Question:

In the cell below, summarize the findings of this experiment, and WHY these results make sense.

Answer:

The batchnorm method is much more robust to weight initialization scales. The baseline model shows a clear convergence only for a small range of weight initialization around 10^-1, while the batchnorm performs reasonably well for the whole range. Batchnormalization also seems to prevent the explosion of loss value around 1, while the baseline method goes far off the plot.

The intuition behind this is that batchnorm prevents the cascading effects of weights that are too small or too large. In the baseline, if the weights are initialized at too small of a value, the activations become smaller and smaller with each layer, resulting in a lot of dead neurons. If the weights are too large, the activations become larger with each layer, resulting in extremely high values towards the end of the net, which causes the loss explosion. In the batchnorm implementation, these values are normalized between each layer, preventing the cascade effect.

Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive over 60% accuracy on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, and their layer structure. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

In [1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from nndl.layers import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numer
ical_gradient_array
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-module
s-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 [2]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

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

Dropout forward pass

Implement the training and test time dropout forward pass, `dropout_forward`, in `nndl/layers.py`. After that, test your implementation by running the following cell.

In [3]:
```
x = np.random.randn(500, 500) + 10

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

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

```
('Running tests with p = ', 0.3)
('Mean of input: ', 10.001082235485809)
('Mean of train-time output: ', 9.999702017393306)
('Mean of test-time output: ', 10.001082235485809)
('Fraction of train-time output set to zero: ', 0.300028)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.6)
('Mean of input: ', 10.001082235485809)
('Mean of train-time output: ', 10.006773002777756)
('Mean of test-time output: ', 10.001082235485809)
('Fraction of train-time output set to zero: ', 0.599804)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.75)
('Mean of input: ', 10.001082235485809)
('Mean of train-time output: ', 10.012936754081702)
('Mean of test-time output: ', 10.001082235485809)
('Fraction of train-time output set to zero: ', 0.749844)
('Fraction of test-time output set to zero: ', 0.0)
```

Dropout backward pass

Implement the backward pass, `dropout_backward`, in `nndl/layers.py`. After that, test your gradients by running the following cell:

In [4]:
```
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, '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], x, dout)

print('dx relative error: ', rel_error(dx, dx_num))
```

```
('dx relative error: ', 1.892903972810276e-11)
```

Implement a fully connected neural network with dropout layers

Modify the `FullyConnectedNet()` class in `nndl/fc_net.py` to incorporate dropout. A dropout layer should be incorporated after every ReLU layer. Concretely, there shouldn't be a dropout at the output layer since there is no ReLU at the output layer. You will need to modify the class in the following areas:

(1) In the forward pass, you will need to incorporate a dropout layer after every relu layer.

(2) In the backward pass, you will need to incorporate a dropout backward pass layer.

Check your implementation by running the following code. Our W1 gradient relative error is on the order of 1e-6 (the largest of all the relative errors).

In [5]:
```
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 [0, 0.25, 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)

  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('{} relative error: {}'.format(name, rel_error(grad_num, gr
ads[name])))
  print('\n')
```

```
('Running check with dropout = ', 0)
('Initial loss: ', 2.3051948273987857)
W1 relative error: 5.25426264222e-07
W2 relative error: 1.88756029969e-05
W3 relative error: 2.91609738888e-07
b1 relative error: 1.34135249104e-07
b2 relative error: 7.09286957083e-08
b3 relative error: 1.4926760615e-10


('Running check with dropout = ', 0.25)
('Initial loss: ', 2.3052077546540826)
W1 relative error: 2.61384694481e-07
W2 relative error: 1.00340102207e-07
W3 relative error: 4.45631607704e-08
b1 relative error: 1.79278481749e-07
b2 relative error: 5.03584968497e-09
b3 relative error: 1.00397473212e-10


('Running check with dropout = ', 0.5)
('Initial loss: ', 2.3035667586595423)
W1 relative error: 1.93342115799e-06
W2 relative error: 7.42499917861e-08
W3 relative error: 7.40458236465e-09
b1 relative error: 7.42143754193e-08
b2 relative error: 4.4872977417e-10
b3 relative error: 1.45584710338e-10
```

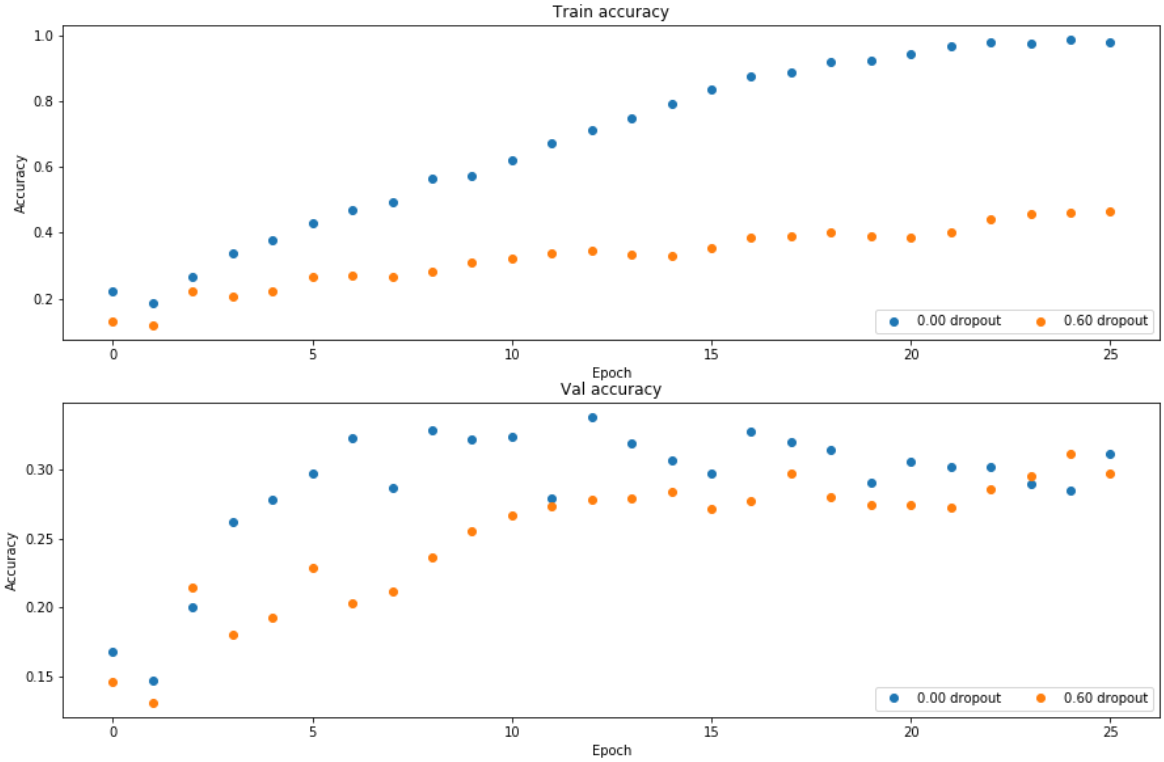
Dropout as a regularizer

In class, we claimed that dropout acts as a regularizer by effectively bagging. To check this, we will train two small networks, one with dropout and one without dropout.

In [6]:
```
# Train two identical nets, one with dropout and one without

num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [0, 0.6]
for dropout in dropout_choices:
  model = FullyConnectedNet([100, 100, 100], dropout=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
```

```
(Iteration 1 / 125) loss: 2.300804
(Epoch 0 / 25) train acc: 0.220000; val_acc: 0.168000
(Epoch 1 / 25) train acc: 0.188000; val_acc: 0.147000
(Epoch 2 / 25) train acc: 0.266000; val_acc: 0.200000
(Epoch 3 / 25) train acc: 0.338000; val_acc: 0.262000
(Epoch 4 / 25) train acc: 0.378000; val_acc: 0.278000
(Epoch 5 / 25) train acc: 0.428000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.468000; val_acc: 0.323000
(Epoch 7 / 25) train acc: 0.494000; val_acc: 0.287000
(Epoch 8 / 25) train acc: 0.566000; val_acc: 0.328000
(Epoch 9 / 25) train acc: 0.572000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.622000; val_acc: 0.324000
(Epoch 11 / 25) train acc: 0.670000; val_acc: 0.279000
(Epoch 12 / 25) train acc: 0.710000; val_acc: 0.338000
(Epoch 13 / 25) train acc: 0.746000; val_acc: 0.319000
(Epoch 14 / 25) train acc: 0.792000; val_acc: 0.307000
(Epoch 15 / 25) train acc: 0.834000; val_acc: 0.297000
(Epoch 16 / 25) train acc: 0.876000; val_acc: 0.327000
(Epoch 17 / 25) train acc: 0.886000; val_acc: 0.320000
(Epoch 18 / 25) train acc: 0.918000; val_acc: 0.314000
(Epoch 19 / 25) train acc: 0.922000; val_acc: 0.290000
(Epoch 20 / 25) train acc: 0.944000; val_acc: 0.306000
(Iteration 101 / 125) loss: 0.156105
(Epoch 21 / 25) train acc: 0.968000; val_acc: 0.302000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.302000
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.986000; val_acc: 0.285000
(Epoch 25 / 25) train acc: 0.978000; val_acc: 0.311000
(Iteration 1 / 125) loss: 2.298716
(Epoch 0 / 25) train acc: 0.132000; val_acc: 0.146000
(Epoch 1 / 25) train acc: 0.118000; val_acc: 0.131000
(Epoch 2 / 25) train acc: 0.220000; val_acc: 0.214000
(Epoch 3 / 25) train acc: 0.206000; val_acc: 0.180000
(Epoch 4 / 25) train acc: 0.220000; val_acc: 0.193000
(Epoch 5 / 25) train acc: 0.264000; val_acc: 0.229000
(Epoch 6 / 25) train acc: 0.268000; val_acc: 0.203000
(Epoch 7 / 25) train acc: 0.266000; val_acc: 0.212000
(Epoch 8 / 25) train acc: 0.282000; val_acc: 0.236000
(Epoch 9 / 25) train acc: 0.310000; val_acc: 0.255000
(Epoch 10 / 25) train acc: 0.320000; val_acc: 0.267000
(Epoch 11 / 25) train acc: 0.338000; val_acc: 0.273000
(Epoch 12 / 25) train acc: 0.346000; val_acc: 0.278000
(Epoch 13 / 25) train acc: 0.332000; val_acc: 0.279000
(Epoch 14 / 25) train acc: 0.328000; val_acc: 0.284000
(Epoch 15 / 25) train acc: 0.354000; val_acc: 0.271000
(Epoch 16 / 25) train acc: 0.386000; val_acc: 0.277000
(Epoch 17 / 25) train acc: 0.388000; val_acc: 0.297000
(Epoch 18 / 25) train acc: 0.402000; val_acc: 0.280000
(Epoch 19 / 25) train acc: 0.388000; val_acc: 0.274000
(Epoch 20 / 25) train acc: 0.386000; val_acc: 0.274000
(Iteration 101 / 125) loss: 1.919649
(Epoch 21 / 25) train acc: 0.402000; val_acc: 0.272000
(Epoch 22 / 25) train acc: 0.440000; val_acc: 0.286000
(Epoch 23 / 25) train acc: 0.458000; val_acc: 0.295000
(Epoch 24 / 25) train acc: 0.462000; val_acc: 0.311000
(Epoch 25 / 25) train acc: 0.466000; val_acc: 0.297000
```

In [7]:
```
# 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 dropo
ut' % 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 dropou
t' % 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()
```



Question

Based off the results of this experiment, is dropout performing regularization? Explain your answer.

Answer:

By the plots, it looks like it is. You can see that the training accuracy gets very far ahead of the validation accuracy with 0 dropout, suggesting the model is overfitting. With .6 dropout, the validation accuracy and training accuracy are more or less in step.

Final part of the assignment

Get over 60% validation accuracy on CIFAR-10 by using the layers you have implemented. You will be graded according to the following equation:

min(floor((X - 32%)) / 28%, 1) where if you get 60% or higher validation accuracy, you get full points.

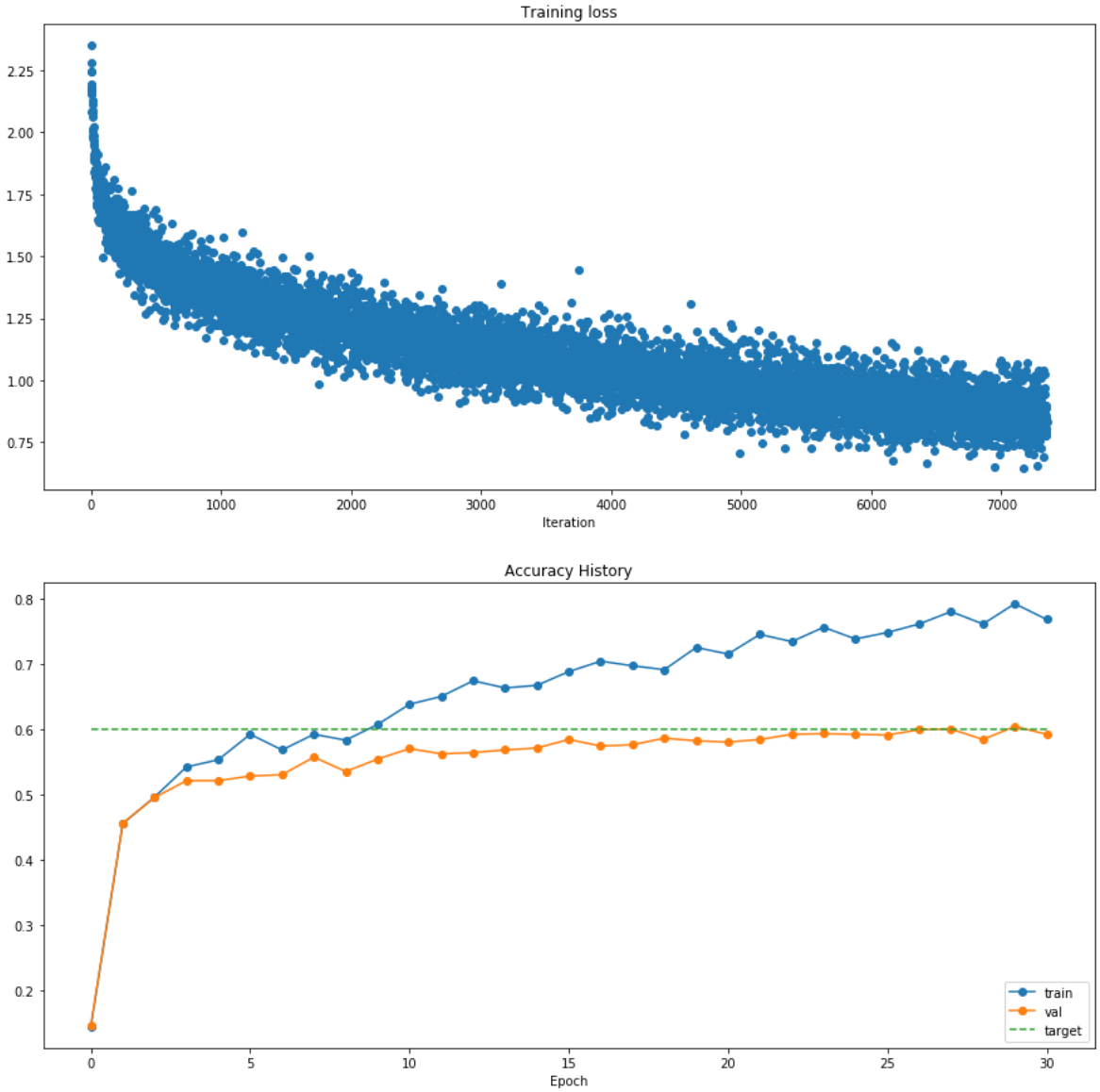
In [11]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #


hidden_dims = [500, 500, 500, 500]
learning_rates = [1e-2, 1e-3, 1e-4]
optimizer = ['adam', 'sgd_nesterov_momentum']
weight_scale = [1e-2, 1e-3]
lr_decay = [.9, .95]
bs  = 200
epochs = 30
dropout = [.2, .3]
use_bn = ['True', 'False']

best_val_acc = -1
best_params = None
best_settings = None

"""

for lr in learning_rates:
    for op in optimizer:
        for ws in weight_scale:
            for dec in lr_decay:
                for do in dropout:
                    for bn in use_bn:

                    model = FullyConnectedNet(hidden_dims, dropout=d
o,
```

```
                                                weight_scale=ws, use_bat
chnorm = bn)

                    solver = Solver(model, data,
                                    num_epochs=epochs, batch_size =bs,
                                    update_rule = op,
                                    optim_config= {
                                        'learning_rate' : lr
                                    },
                                    lr_decay=dec,
                                    print_every=400)

                    solver.train()


                    print solver.best_val_acc

                    if solver.best_val_acc > best_val_acc:
                        best_val_acc = solver.best_val_acc
                        best_params = solver.best_params
                        best_settings = (lr, op, ws, dec, do)
                        best_model = model









"""



model = FullyConnectedNet(hidden_dims, dropout=.35,
                          weight_scale=1e-2, use_batchnorm=True)

solver = Solver(model, data,
                num_epochs=epochs, batch_size=bs,
                update_rule = 'adam',
                optim_config= {
                    'learning_rate' : 5e-4
                },
                lr_decay=.95,
                print_every=400)

time_start = time.time()

solver.train()

print 'Training time: {}'.format(time.time()-time_start)




# ============================================================ #
# END YOUR CODE HERE
# ============================================================ #
```

```
(Iteration 1 / 7350) loss: 2.349134
(Epoch 0 / 30) train acc: 0.144000; val_acc: 0.146000
(Epoch 1 / 30) train acc: 0.455000; val_acc: 0.456000
(Iteration 401 / 7350) loss: 1.511353
(Epoch 2 / 30) train acc: 0.496000; val_acc: 0.495000
(Epoch 3 / 30) train acc: 0.542000; val_acc: 0.521000
(Iteration 801 / 7350) loss: 1.495701
(Epoch 4 / 30) train acc: 0.553000; val_acc: 0.521000
(Iteration 1201 / 7350) loss: 1.305079
(Epoch 5 / 30) train acc: 0.592000; val_acc: 0.528000
(Epoch 6 / 30) train acc: 0.568000; val_acc: 0.530000
(Iteration 1601 / 7350) loss: 1.346184
(Epoch 7 / 30) train acc: 0.592000; val_acc: 0.557000
(Epoch 8 / 30) train acc: 0.583000; val_acc: 0.535000
(Iteration 2001 / 7350) loss: 1.166619
(Epoch 9 / 30) train acc: 0.607000; val_acc: 0.554000
(Iteration 2401 / 7350) loss: 1.082266
(Epoch 10 / 30) train acc: 0.638000; val_acc: 0.570000
(Epoch 11 / 30) train acc: 0.650000; val_acc: 0.562000
(Iteration 2801 / 7350) loss: 1.148503
(Epoch 12 / 30) train acc: 0.674000; val_acc: 0.564000
(Epoch 13 / 30) train acc: 0.663000; val_acc: 0.568000
(Iteration 3201 / 7350) loss: 1.081492
(Epoch 14 / 30) train acc: 0.667000; val_acc: 0.571000
(Iteration 3601 / 7350) loss: 1.148346
(Epoch 15 / 30) train acc: 0.688000; val_acc: 0.584000
(Epoch 16 / 30) train acc: 0.704000; val_acc: 0.574000
(Iteration 4001 / 7350) loss: 1.077373
(Epoch 17 / 30) train acc: 0.697000; val_acc: 0.576000
(Iteration 4401 / 7350) loss: 0.881307
(Epoch 18 / 30) train acc: 0.691000; val_acc: 0.586000
(Epoch 19 / 30) train acc: 0.725000; val_acc: 0.582000
(Iteration 4801 / 7350) loss: 0.826899
(Epoch 20 / 30) train acc: 0.715000; val_acc: 0.580000
(Epoch 21 / 30) train acc: 0.745000; val_acc: 0.584000
(Iteration 5201 / 7350) loss: 0.943966
(Epoch 22 / 30) train acc: 0.734000; val_acc: 0.592000
(Iteration 5601 / 7350) loss: 0.772600
(Epoch 23 / 30) train acc: 0.756000; val_acc: 0.593000
(Epoch 24 / 30) train acc: 0.738000; val_acc: 0.592000
(Iteration 6001 / 7350) loss: 0.953089
(Epoch 25 / 30) train acc: 0.748000; val_acc: 0.591000
(Epoch 26 / 30) train acc: 0.761000; val_acc: 0.599000
(Iteration 6401 / 7350) loss: 0.984805
(Epoch 27 / 30) train acc: 0.780000; val_acc: 0.600000
(Iteration 6801 / 7350) loss: 0.946586
(Epoch 28 / 30) train acc: 0.761000; val_acc: 0.584000
(Epoch 29 / 30) train acc: 0.792000; val_acc: 0.604000
(Iteration 7201 / 7350) loss: 0.825021
(Epoch 30 / 30) train acc: 0.768000; val_acc: 0.592000
Training time: 1063.93349099
```

In [12]:
```
print solver.best_val_acc
plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
plt.plot(solver.loss_history, 'o')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot(.6*np.ones_like(solver.val_acc_history), '--', label='targe
t')
plt.title('Accuracy History')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
0.604
```



```python
import numpy as np
import pdb

from .layers import *
from .layer_utils import *

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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

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

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

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

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

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

    # ============================================================ #
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params dictionary.
    #   The weights and biases of layer 1 are W1 and b1; and in general the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #
    #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
    #   parameters to zero.  The gamma and beta parameters for layer 1 should
    #   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
    #   should be gamma2 and beta2, etc. Only use batchnorm if self.use_batchnorm
    #   is true and DO NOT batch normalize the output scores.
    # ============================================================ #
    dimensions = [input_dim] + hidden_dims + [num_classes]

    for i in np.arange(self.num_layers):
      self.params['W{}'.format(i+1)] = weight_scale * np.random.randn(dimensions[i], dimensions[i+1])
      self.params['b{}'.format(i+1)] = np.zeros(dimensions[i+1])
      if self.use_batchnorm and i < self.num_layers-1:
        self.params['gamma{}'.format(i+1)] = np.ones((1,dimensions[i+1]))
        self.params['beta{}'.format(i+1)] = np.zeros((1,dimensions[i+1]))



    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

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

```

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

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


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

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

    # Set train/test mode for batchnorm params and dropout param since they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    #
    #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
    #   between the affine_forward and relu_forward layers.  You may
    #   also write an affine_batchnorm_relu() function in layer_utils.py.
    #
    #   DROPOUT: If dropout is non-zero, insert a dropout layer after
    #   every ReLU layer.
    # ============================================================ #
    caches = []
    layer_scores = []
    do_caches = []
    do_cache = None

    layer_scores.append(X)
    temp_score = X

    for i in np.arange(self.num_layers-1):
      if self.use_batchnorm:
        temp_score, temp_cache = affine_batchnorm_relu(temp_score, self.params['W{}'.format(i+1)],
          self.params['b{}'.format(i+1)], self.params['gamma{}'.format(i+1)],
          self.params['beta{}'.format(i+1)], self.bn_params[i])
      else:
        temp_score, temp_cache = affine_relu_forward(temp_score, self.params['W{}'.format(i+1)],
          self.params['b{}'.format(i+1)])
      caches.append(temp_cache)
      layer_scores.append(temp_score)
      if self.use_dropout:
        temp_score, do_cache = dropout_forward(temp_score, self.dropout_param)
        do_caches.append(do_cache)




    temp_score, temp_cache = affine_forward(temp_score, self.params['W{}'.format(self.num_layers)], self.params['b{}'.format(self.num_layers)])
    caches.append(temp_cache)
    layer_scores.append(temp_score)

    scores = layer_scores[-1]

    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

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

    loss, grads = 0.0, {}
    # ============================================================ #
    # YOUR CODE HERE:
    #   Implement the backwards pass of the FC net and store the gradients
    #   in the grads dict, so that grads[k] is the gradient of self.params[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    #
    #   BATCHNORM: Incorporate the backward pass of the batchnorm.
    #
    #   DROPOUT: Incorporate the backward pass of dropout.
    # ============================================================ #
    num_examples = scores.shape[0]

    max_score = np.amax(scores, axis=1)
    scores -= max_score[:, np.newaxis]

    e_scores = np.exp(scores)
```

```
202     sums = np.sum(e_scores, axis=1)
203     log_sums = np.log(sums)
204     y_terms = scores[np.arange(num_examples), y]
205
206     reg_loss = 0
207     for i in np.arange(self.num_layers):
208       W = self.params['W{}'.format(i+1)]
209       reg_loss += .5*self.reg*np.sum(W*W)
210
211     loss = np.sum(log_sums - y_terms)/num_examples + reg_loss
212
213     d_scores = e_scores/sums[:,np.newaxis]
214     d_scores[np.arange(num_examples),y] -= 1
215     d_scores = d_scores/num_examples
216
217     dx, dw, db = affine_backward(d_scores, caches[self.num_layers-1])
218     grads['W{}'.format(self.num_layers)] = dw + self.reg*self.params['W{}'.format(self.num_layers)]
219     grads['b{}'.format(self.num_layers)] = db
220
221     for i in np.arange(self.num_layers-2, -1,-1):
222       if self.use_dropout:
223         dx = dropout_backward(dx, do_caches[i])
224       if self.use_batchnorm:
225         dx, dw, db, dgamma, dbeta = affine_batchnorm_relu_back (dx, caches[i])
226         grads['gamma{}'.format(i+1)] = dgamma
227         grads['beta{}'.format(i+1)] = dbeta
228       else:
229         dx, dw, db = affine_relu_backward(dx, caches[i])
230       grads['W{}'.format(i+1)] = dw + self.reg*self.params['W{}'.format(i+1)]
231       grads['b{}'.format(i+1)] = db
232
233
234
235     # ============================================================ #
236     # END YOUR CODE HERE
237     # ============================================================ #
238
239     return loss, grads
```

```python
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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

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

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

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ============================================================ #

  shape = x.shape
  N = shape[0]
  D = np.prod(shape[1:])
  reshaped_x = np.reshape(x, (N,D))

  out = reshaped_x.dot(w) + b[:, np.newaxis].T

  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #

  cache = (x, w, b)
  return out, cache


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

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

  Returns a tuple of:
  - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
  - dw: Gradient with respect to w, of shape (D, M)
  - db: Gradient with respect to b, of shape (M,)
  """
```

```
  70   x, w, b = cache
  71   dx, dw, db = None, None, None
  72
  73   # ============================================================== #
  74   # YOUR CODE HERE:
  75   #   Calculate the gradients for the backward pass.
  76   # ============================================================== #
  77
  78   N, M = dout.shape
  79   D = w.shape[0]
  80   reshaped_x = np.reshape(x, (N,D))
  81
  82   db = np.sum(dout, axis=0)
  83   dw = reshaped_x.T.dot(dout)
  84   dx = np.reshape(dout.dot(w.T), x.shape)
  85
  86   # ============================================================== #
  87   # END YOUR CODE HERE
  88   # ============================================================== #
  89
  90   return dx, dw, db
  91
  92 def relu_forward(x):
  93   """
  94   Computes the forward pass for a layer of rectified linear units (ReLUs).
  95
  96   Input:
  97   - x: Inputs, of any shape
  98
  99   Returns a tuple of:
 100   - out: Output, of the same shape as x
 101   - cache: x
 102   """
 103   # ============================================================== #
 104   # YOUR CODE HERE:
 105   #   Implement the ReLU forward pass.
 106   # ============================================================== #
 107   out = np.empty_like(x)
 108   out[:] = x
 109   out[out<0] = 0
 110   # ============================================================== #
 111   # END YOUR CODE HERE
 112   # ============================================================== #
 113
 114   cache = x
 115   return out, cache
 116
 117
 118 def relu_backward(dout, cache):
 119   """
 120   Computes the backward pass for a layer of rectified linear units (ReLUs).
 121
 122   Input:
 123   - dout: Upstream derivatives, of any shape
 124   - cache: Input x, of same shape as dout
 125
 126   Returns:
 127   - dx: Gradient with respect to x
 128   """
 129   x = cache
 130
 131   # ============================================================== #
 132   # YOUR CODE HERE:
 133   #   Implement the ReLU backward pass
 134   # ============================================================== #
 135
 136   dx = np.empty_like(dout)
 137   dx[:] = dout
 138   dx[x<0] = 0
```

```
139
140   # ============================================================ #
141   # END YOUR CODE HERE
142   # ============================================================ #
143
144   return dx
145
146 def batchnorm_forward(x, gamma, beta, bn_param):
147   """
148   Forward pass for batch normalization.
149
150   During training the sample mean and (uncorrected) sample variance are
151   computed from minibatch statistics and used to normalize the incoming data.
152   During training we also keep an exponentially decaying running mean of the mean
153   and variance of each feature, and these averages are used to normalize data
154   at test-time.
155
156   At each timestep we update the running averages for mean and variance using
157   an exponential decay based on the momentum parameter:
158
159   running_mean = momentum * running_mean + (1 - momentum) * sample_mean
160   running_var = momentum * running_var + (1 - momentum) * sample_var
161
162   Note that the batch normalization paper suggests a different test-time
163   behavior: they compute sample mean and variance for each feature using a
164   large number of training images rather than using a running average. For
165   this implementation we have chosen to use running averages instead since
166   they do not require an additional estimation step; the torch7 implementation
167   of batch normalization also uses running averages.
168
169   Input:
170   - x: Data of shape (N, D)
171   - gamma: Scale parameter of shape (D,)
172   - beta: Shift paremeter of shape (D,)
173   - bn_param: Dictionary with the following keys:
174     - mode: 'train' or 'test'; required
175     - eps: Constant for numeric stability
176     - momentum: Constant for running mean / variance.
177     - running_mean: Array of shape (D,) giving running mean of features
178     - running_var Array of shape (D,) giving running variance of features
179
180   Returns a tuple of:
181   - out: of shape (N, D)
182   - cache: A tuple of values needed in the backward pass
183   """
184   mode = bn_param['mode']
185   eps = bn_param.get('eps', 1e-5)
186   momentum = bn_param.get('momentum', 0.9)
187
188   N, D = x.shape
189   running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
190   running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
191
192   out, cache = None, None
193   if mode == 'train':
194
195     # ============================================================ #
196     # YOUR CODE HERE:
197     #   A few steps here:
198     #     (1) Calculate the running mean and variance of the minibatch.
199     #     (2) Normalize the activations with the running mean and variance.
200     #     (3) Scale and shift the normalized activations.  Store this
201     #         as the variable 'out'
202     #     (4) Store any variables you may need for the backward pass in
203     #         the 'cache' variable.
204     # ============================================================ #
205
206     sample_mean = x.mean(axis=0)
207     sample_var = x.var(axis=0)
```

```
208
209     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
210     running_var = momentum * running_var + (1 - momentum) * sample_var
211
212     x_hat = x - sample_mean[:, np.newaxis].T
213     x_hat /= np.sqrt((sample_var[:, np.newaxis].T + eps))
214
215     out = gamma*x_hat + beta
216
217     cache = (x, gamma, x_hat, sample_var, sample_mean, eps)
218
219     # ============================================================ #
220     # END YOUR CODE HERE
221     # ============================================================ #
222
223   elif mode == 'test':
224
225     # ============================================================ #
226     # YOUR CODE HERE:
227     #   Calculate the testing time normalized activations.  Normalize using
228     #   the running mean and variance, and then scale and shift appropriately.
229     #   Store the output as 'out'.
230     # ============================================================ #
231
232     out = x - running_mean[:, np.newaxis].T
233     out /= np.sqrt((running_var[:, np.newaxis].T + eps))
234
235     out = gamma*out + beta
236
237
238     # ============================================================ #
239     # END YOUR CODE HERE
240     # ============================================================ #
241
242   else:
243     raise ValueError('Invalid forward batchnorm mode " %s"' % mode)
244
245   # Store the updated running means back into bn_param
246   bn_param['running_mean'] = running_mean
247   bn_param['running_var'] = running_var
248
249   return out, cache
250
251 def batchnorm_backward(dout, cache):
252   """
253   Backward pass for batch normalization.
254
255   For this implementation, you should write out a computation graph for
256   batch normalization on paper and propagate gradients backward through
257   intermediate nodes.
258
259   Inputs:
260   - dout: Upstream derivatives, of shape (N, D)
261   - cache: Variable of intermediates from batchnorm_forward.
262
263   Returns a tuple of:
264   - dx: Gradient with respect to inputs x, of shape (N, D)
265   - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
266   - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
267   """
268   dx, dgamma, dbeta = None, None, None
269
270   # ============================================================ #
271   # YOUR CODE HERE:
272   #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
273   # ============================================================ #
274   x, gamma, x_hat, var, mu, eps = cache
275
276   N = x.shape[0]
```

```
277
278   dgamma = np.sum(np.multiply(x_hat, dout), axis=0)
279   dbeta = dout.sum(axis=0)
280
281   dx_hat = np.multiply(gamma, dout)
282
283   dmu = -1.0/np.sqrt(var + eps)*np.sum(dx_hat, axis=0)
284   da = np.multiply(1/np.sqrt(var+eps), dx_hat)
285   de = -.5*1.0/np.power(var+eps, 1.5)*(x - mu)*dx_hat
286   dvar = np.sum(de, axis=0)
287
288   dx = da + 2.0*(x-mu)/N*dvar + 1.0/N*dmu
289
290
291
292
293   # ============================================================== #
294   # END YOUR CODE HERE
295   # ============================================================== #
296
297   return dx, dgamma, dbeta
298
299 def dropout_forward(x, dropout_param):
300   """
301   Performs the forward pass for (inverted) dropout.
302
303   Inputs:
304   - x: Input data, of any shape
305   - dropout_param: A dictionary with the following keys:
306     - p: Dropout parameter. We drop each neuron output with probability p.
307     - mode: 'test' or 'train'. If the mode is train, then perform dropout;
308       if the mode is test, then just return the input.
309     - seed: Seed for the random number generator. Passing seed makes this
310       function deterministic, which is needed for gradient checking but not in
311       real networks.
312
313   Outputs:
314   - out: Array of the same shape as x.
315   - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
316     mask that was used to multiply the input; in test mode, mask is None.
317   """
318   p, mode = dropout_param['p'], dropout_param['mode']
319   if 'seed' in dropout_param:
320     np.random.seed(dropout_param['seed'])
321
322   mask = None
323   out = None
324
325   if mode == 'train':
326     # ============================================================== #
327     # YOUR CODE HERE:
328     #   Implement the inverted dropout forward pass during training time.
329     #   Store the masked and scaled activations in out, and store the
330     #   dropout mask as the variable mask.
331     # ============================================================== #
332
333     mask = (np.random.rand(*x.shape) > p)/(1-p)
334     out = x*mask
335
336     # ============================================================== #
337     # END YOUR CODE HERE
338     # ============================================================== #
339
340   elif mode == 'test':
341
342     # ============================================================== #
343     # YOUR CODE HERE:
344     #   Implement the inverted dropout forward pass during test time.
345     # ============================================================== #
```

```
346
347     out = np.empty_like(x)
348     out[:] = x
349
350
351     # ============================================================================ #
352     # END YOUR CODE HERE
353     # ============================================================================ #
354
355   cache = (dropout_param, mask)
356   out = out.astype(x.dtype, copy=False)
357
358   return out, cache
359
360 def dropout_backward(dout, cache):
361   """
362   Perform the backward pass for (inverted) dropout.
363
364   Inputs:
365   - dout: Upstream derivatives, of any shape
366   - cache: (dropout_param, mask) from dropout_forward.
367   """
368   dropout_param, mask = cache
369   mode = dropout_param['mode']
370
371   dx = None
372   if mode == 'train':
373     # ============================================================================ #
374     # YOUR CODE HERE:
375     #   Implement the inverted dropout backward pass during training time.
376     # ============================================================================ #
377     dx = dout*mask
378     # ============================================================================ #
379     # END YOUR CODE HERE
380     # ============================================================================ #
381   elif mode == 'test':
382     # ============================================================================ #
383     # YOUR CODE HERE:
384     #   Implement the inverted dropout backward pass during test time.
385     # ============================================================================ #
386     dx = np.empty_like(dout)
387     dx[:] = dout
388     # ============================================================================ #
389     # END YOUR CODE HERE
390     # ============================================================================ #
391   return dx
392
393 def svm_loss(x, y):
394   """
395   Computes the loss and gradient using for multiclass SVM classification.
396
397   Inputs:
398   - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
399     for the ith input.
400   - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
401     0 <= y[i] < C
402
403   Returns a tuple of:
404   - loss: Scalar giving the loss
405   - dx: Gradient of the loss with respect to x
406   """
407   N = x.shape[0]
408   correct_class_scores = x[np.arange(N), y]
409   margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
410   margins[np.arange(N), y] = 0
411   loss = np.sum(margins) / N
412   num_pos = np.sum(margins > 0, axis=1)
413   dx = np.zeros_like(x)
414   dx[margins > 0] = 1
```

```
415   dx[np.arange(N), y] -= num_pos
416   dx /= N
417   return loss, dx
418
419
420 def softmax_loss(x, y):
421   """
422   Computes the loss and gradient for softmax classification.
423
424   Inputs:
425   - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
426     for the ith input.
427   - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
428     0 <= y[i] < C
429
430   Returns a tuple of:
431   - loss: Scalar giving the loss
432   - dx: Gradient of the loss with respect to x
433   """
434
435   probs = np.exp(x - np.max(x, axis=1, keepdims=True))
436   probs /= np.sum(probs, axis=1, keepdims=True)
437   N = x.shape[0]
438   loss = -np.sum(np.log(probs[np.arange(N), y])) / N
439   dx = probs.copy()
440   dx[np.arange(N), y] -= 1
441   dx /= N
442   return loss, dx
```

```python
from .layers import *

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

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

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

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


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



def affine_batchnorm_relu(x, w, b, gamma, beta, bn_param):
  a, fc_cache = affine_forward(x, w, b)
  a1, bn_cache = batchnorm_forward(a, gamma, beta, bn_param)
  out, relu_cache = relu_forward(a1)
  cache = (fc_cache, bn_cache, relu_cache)
  return out, cache


def affine_batchnorm_relu_back(dout, cache):
  fc_cache, bn_cache, relu_cache = cache
  da = relu_backward(dout, relu_cache)
  da1, dgamma, dbeta = batchnorm_backward(da, bn_cache)
  dx, dw, db = affine_backward(da1, fc_cache)
  return dx, dw, db, dgamma, dbeta
```