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 [4]:
```
## 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_numerica
l_gradient_array
from cs231n.solver import Solver

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

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

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

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

In [5]:
```
# 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 [6]:
```
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 1
e-9:
difference: 9.76985004799e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 2.49531659997e-10
dw error: 8.61597493627e-11
db error: 1.07547050157e-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: 3.2756027939e-12


If affine_relu_forward and affine_relu_backward are working, error shou
ld be less than 1e-9::
dx error: 8.99810200276e-11
dw error: 3.79096661278e-10
db error: 7.8267287404e-12


Running check with reg = 0
Initial loss: 2.30489194644
W1 relative error: 1.76344222058e-07
W2 relative error: 1.95676842808e-06
W3 relative error: 7.12979384004e-07
b1 relative error: 3.11799412167e-08
b2 relative error: 2.51208563234e-09
b3 relative error: 1.01159858345e-10
Running check with reg = 3.14
Initial loss: 6.65152842538
W1 relative error: 6.64550089788e-08
W2 relative error: 2.92270057456e-08
W3 relative error: 2.2894570596e-08
b1 relative error: 8.25046381658e-08
b2 relative error: 2.03699864699e-08
b3 relative error: 3.50848933861e-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 [7]:
```
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['v
elocity'])))
```

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

SGD + Nesterov momentum

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

In [8]:
```
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['v
elocity'])))
```

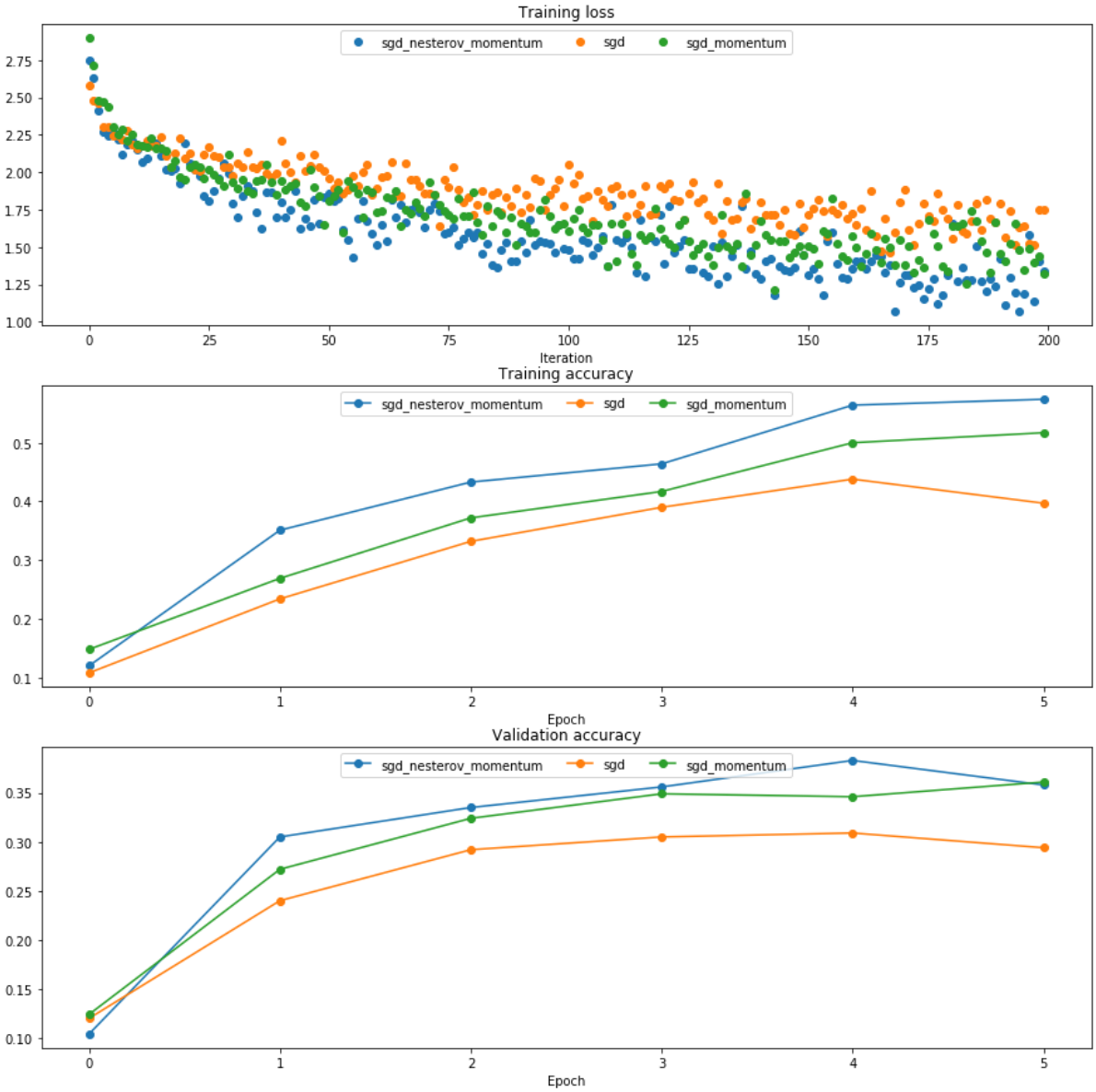
```
next_w error: 1.08751868451e-08
velocity error: 4.26928774328e-09
```

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 [9]:
```
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=5e-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
```



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 [11]:
```
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: 9.50264522989e-08
cache error: 2.64779558072e-09
```

Adaptive moments

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

In [14]:
```
# 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'])))
```

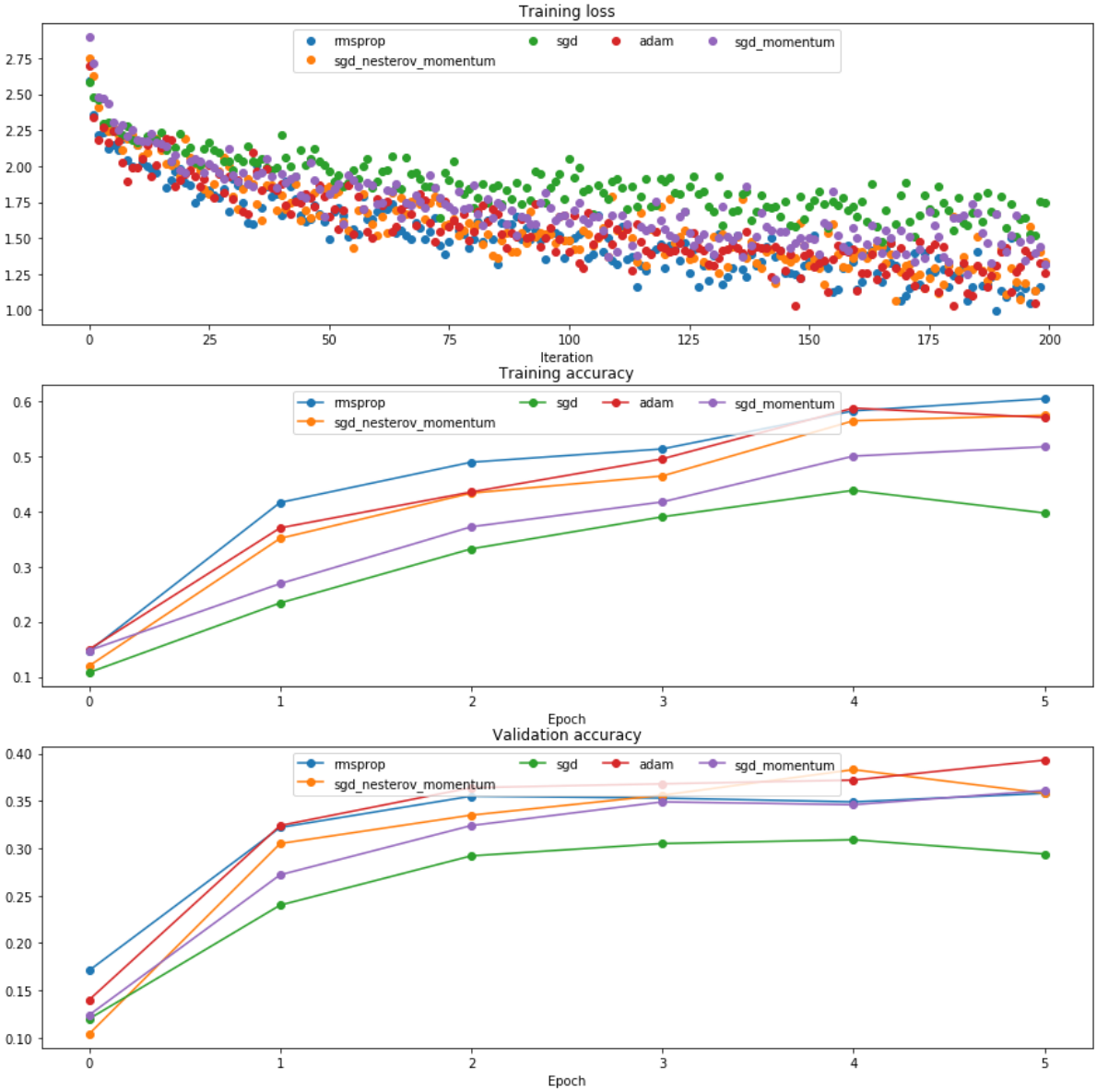
```
next_w error: 1.13956917985e-07
a error: 4.20831403811e-09
v error: 4.21496319311e-09
```

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 [15]:
```
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=5e-2
)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=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 [16]:
```
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: 2.273982
(Epoch 0 / 10) train acc: 0.097000; val_acc: 0.107000
(Iteration 51 / 4900) loss: 1.726994
(Iteration 101 / 4900) loss: 1.825323
(Iteration 151 / 4900) loss: 1.711275
(Iteration 201 / 4900) loss: 1.532059
(Iteration 251 / 4900) loss: 1.658294
(Iteration 301 / 4900) loss: 1.686514
(Iteration 351 / 4900) loss: 1.606527
(Iteration 401 / 4900) loss: 1.554918
(Iteration 451 / 4900) loss: 1.639660
(Epoch 1 / 10) train acc: 0.423000; val_acc: 0.460000
(Iteration 501 / 4900) loss: 1.477871
(Iteration 551 / 4900) loss: 1.579638
(Iteration 601 / 4900) loss: 1.626574
(Iteration 651 / 4900) loss: 1.365260
(Iteration 701 / 4900) loss: 1.590627
(Iteration 751 / 4900) loss: 1.353337
(Iteration 801 / 4900) loss: 1.589906
(Iteration 851 / 4900) loss: 1.486142
(Iteration 901 / 4900) loss: 1.350537
(Iteration 951 / 4900) loss: 1.636652
(Epoch 2 / 10) train acc: 0.495000; val_acc: 0.461000
(Iteration 1001 / 4900) loss: 1.428130
(Iteration 1051 / 4900) loss: 1.493171
(Iteration 1101 / 4900) loss: 1.272455
(Iteration 1151 / 4900) loss: 1.362982
(Iteration 1201 / 4900) loss: 1.395415
(Iteration 1251 / 4900) loss: 1.477466
(Iteration 1301 / 4900) loss: 1.425727
(Iteration 1351 / 4900) loss: 1.475695
(Iteration 1401 / 4900) loss: 1.277570
(Iteration 1451 / 4900) loss: 1.379676
(Epoch 3 / 10) train acc: 0.523000; val_acc: 0.489000
(Iteration 1501 / 4900) loss: 1.512747
(Iteration 1551 / 4900) loss: 1.237747
(Iteration 1601 / 4900) loss: 1.281572
(Iteration 1651 / 4900) loss: 1.400981
(Iteration 1701 / 4900) loss: 1.336948
(Iteration 1751 / 4900) loss: 1.253336
(Iteration 1801 / 4900) loss: 1.168439
(Iteration 1851 / 4900) loss: 1.204696
(Iteration 1901 / 4900) loss: 1.397373
(Iteration 1951 / 4900) loss: 1.235282
(Epoch 4 / 10) train acc: 0.548000; val_acc: 0.480000
(Iteration 2001 / 4900) loss: 1.159441
(Iteration 2051 / 4900) loss: 1.221697
(Iteration 2101 / 4900) loss: 1.152344
(Iteration 2151 / 4900) loss: 1.038060
(Iteration 2201 / 4900) loss: 1.354591
(Iteration 2251 / 4900) loss: 1.244433
(Iteration 2301 / 4900) loss: 1.115347
(Iteration 2351 / 4900) loss: 1.377292
(Iteration 2401 / 4900) loss: 1.234174
(Epoch 5 / 10) train acc: 0.554000; val_acc: 0.529000
(Iteration 2451 / 4900) loss: 1.251489
(Iteration 2501 / 4900) loss: 1.060148
```

```
(Iteration 2551 / 4900) loss: 1.108511
(Iteration 2601 / 4900) loss: 1.126228
(Iteration 2651 / 4900) loss: 1.119353
(Iteration 2701 / 4900) loss: 1.235593
(Iteration 2751 / 4900) loss: 1.065725
(Iteration 2801 / 4900) loss: 1.284755
(Iteration 2851 / 4900) loss: 1.162044
(Iteration 2901 / 4900) loss: 1.331536
(Epoch 6 / 10) train acc: 0.619000; val_acc: 0.525000
(Iteration 2951 / 4900) loss: 1.174324
(Iteration 3001 / 4900) loss: 1.071955
(Iteration 3051 / 4900) loss: 1.235049
(Iteration 3101 / 4900) loss: 1.272271
(Iteration 3151 / 4900) loss: 1.057528
(Iteration 3201 / 4900) loss: 1.120909
(Iteration 3251 / 4900) loss: 1.161273
(Iteration 3301 / 4900) loss: 1.194097
(Iteration 3351 / 4900) loss: 1.023113
(Iteration 3401 / 4900) loss: 1.119645
(Epoch 7 / 10) train acc: 0.631000; val_acc: 0.524000
(Iteration 3451 / 4900) loss: 0.933272
(Iteration 3501 / 4900) loss: 1.233159
(Iteration 3551 / 4900) loss: 1.057111
(Iteration 3601 / 4900) loss: 0.859234
(Iteration 3651 / 4900) loss: 1.129736
(Iteration 3701 / 4900) loss: 1.152087
(Iteration 3751 / 4900) loss: 1.045957
(Iteration 3801 / 4900) loss: 0.888242
(Iteration 3851 / 4900) loss: 1.172544
(Iteration 3901 / 4900) loss: 1.064448
(Epoch 8 / 10) train acc: 0.629000; val_acc: 0.520000
(Iteration 3951 / 4900) loss: 0.998463
(Iteration 4001 / 4900) loss: 0.881860
(Iteration 4051 / 4900) loss: 1.331693
(Iteration 4101 / 4900) loss: 1.010380
(Iteration 4151 / 4900) loss: 1.008343
(Iteration 4201 / 4900) loss: 0.778084
(Iteration 4251 / 4900) loss: 0.963680
(Iteration 4301 / 4900) loss: 1.056844
(Iteration 4351 / 4900) loss: 0.919683
(Iteration 4401 / 4900) loss: 0.954189
(Epoch 9 / 10) train acc: 0.679000; val_acc: 0.544000
(Iteration 4451 / 4900) loss: 1.160176
(Iteration 4501 / 4900) loss: 1.092591
(Iteration 4551 / 4900) loss: 1.076322
(Iteration 4601 / 4900) loss: 0.820000
(Iteration 4651 / 4900) loss: 0.781363
(Iteration 4701 / 4900) loss: 0.893466
(Iteration 4751 / 4900) loss: 0.706025
(Iteration 4801 / 4900) loss: 0.868981
(Iteration 4851 / 4900) loss: 1.044857
(Epoch 10 / 10) train acc: 0.654000; val_acc: 0.529000
```

In [17]:
```
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_tes
t'])))
```

```
Validation set accuracy: 0.544
Test set accuracy: 0.533
```

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_numerica
l_gradient_array
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-i
n-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 variances
# 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': 'tr
ain'})
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([ 8.06197563, 11.7003392 , 17.76882509]))
('  stds: ', array([35.53439411, 36.10063843, 35.37083643]))
After batch normalization (gamma=1, beta=0)
('  mean: ', array([ 1.46549439e-16,  2.10942375e-16, -3.55271368e-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 [5]:
```
# 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([0.05680307, 0.13373348, 0.00947174]))
('  stds: ', array([0.98198094, 1.00873203, 0.9478168 ]))
```

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 [23]:
```
# 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: ', 1.2297161996232763e-08)
('dgamma error: ', 4.282906551077532e-12)
('dbeta error: ', 7.506935052851707e-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 [28]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

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

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

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

```
('Running check with reg = ', 0)
('Initial loss: ', 2.203936542356916)
W1 relative error: 0.000210121845913
W2 relative error: 4.80931315106e-06
W3 relative error: 3.12131730334e-10
b1 relative error: 1.9567680809e-07
b2 relative error: 1.44328993201e-07
b3 relative error: 1.24492152282e-10
beta1 relative error: 1.7866342352e-08
beta2 relative error: 3.07490761247e-09
gamma1 relative error: 1.54874598041e-08
gamma2 relative error: 2.39677653604e-09


('Running check with reg = ', 3.14)
('Initial loss: ', 7.466558659230686)
W1 relative error: 6.69843322195e-05
W2 relative error: 5.6321046722e-06
W3 relative error: 1.93855508471e-08
b1 relative error: 2.84603070277e-09
b2 relative error: 8.32667268469e-08
b3 relative error: 2.46372943163e-10
beta1 relative error: 3.61688183679e-08
beta2 relative error: 4.63376117785e-08
gamma1 relative error: 1.46886316502e-08
gamma2 relative error: 1.64818765589e-08
```

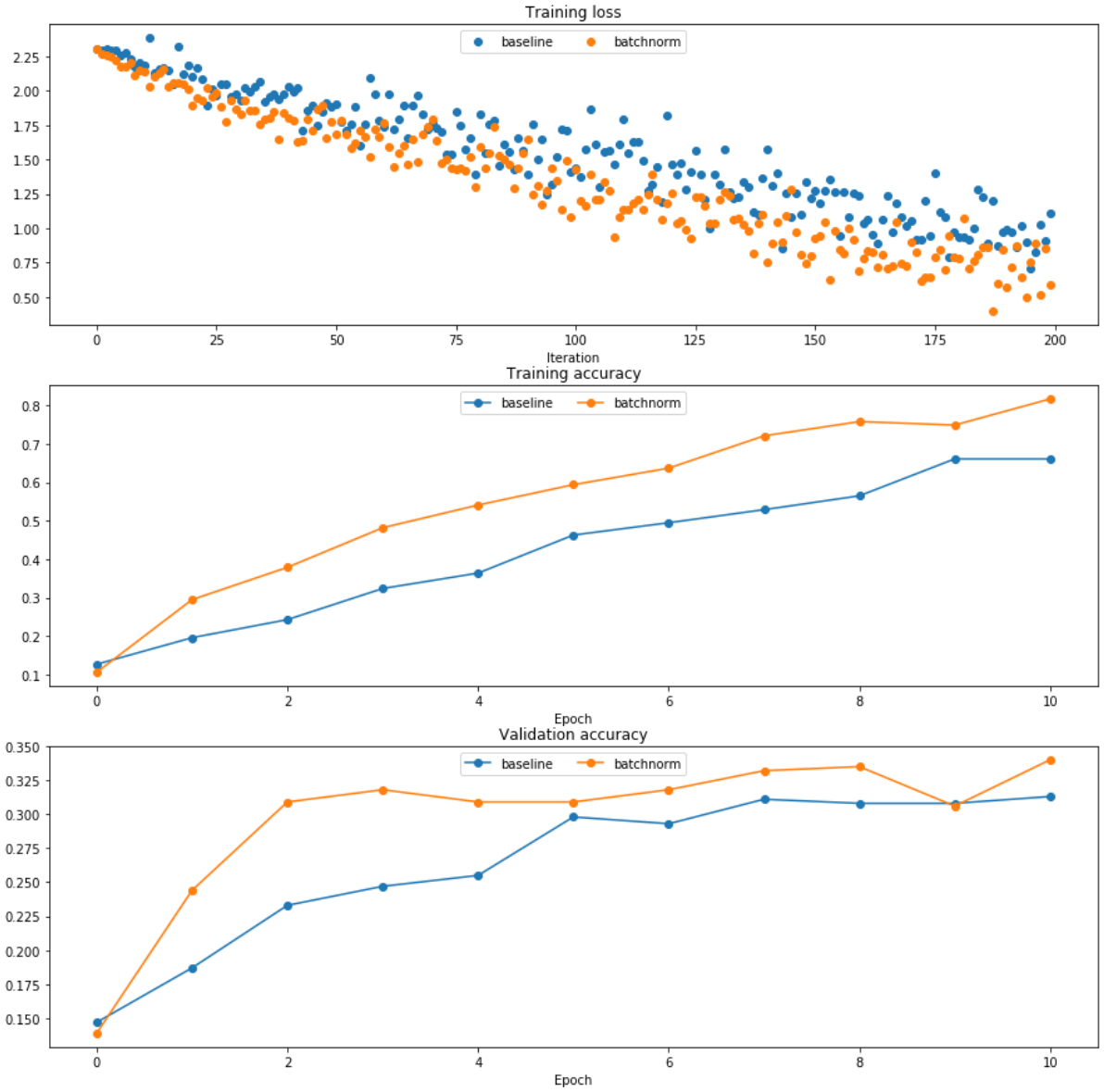
Training a deep fully connected network with batch normalization.

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

In [40]:
```
# 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_ba
tchnorm=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: 2.305009
(Epoch 0 / 10) train acc: 0.106000; val_acc: 0.139000
(Epoch 1 / 10) train acc: 0.295000; val_acc: 0.244000
(Epoch 2 / 10) train acc: 0.379000; val_acc: 0.309000
(Epoch 3 / 10) train acc: 0.482000; val_acc: 0.318000
(Epoch 4 / 10) train acc: 0.541000; val_acc: 0.309000
(Epoch 5 / 10) train acc: 0.594000; val_acc: 0.309000
(Epoch 6 / 10) train acc: 0.637000; val_acc: 0.318000
(Epoch 7 / 10) train acc: 0.721000; val_acc: 0.332000
(Epoch 8 / 10) train acc: 0.758000; val_acc: 0.335000
(Epoch 9 / 10) train acc: 0.749000; val_acc: 0.306000
(Epoch 10 / 10) train acc: 0.817000; val_acc: 0.340000
(Iteration 1 / 200) loss: 2.302242
(Epoch 0 / 10) train acc: 0.127000; val_acc: 0.147000
(Epoch 1 / 10) train acc: 0.196000; val_acc: 0.187000
(Epoch 2 / 10) train acc: 0.243000; val_acc: 0.233000
(Epoch 3 / 10) train acc: 0.324000; val_acc: 0.247000
(Epoch 4 / 10) train acc: 0.364000; val_acc: 0.255000
(Epoch 5 / 10) train acc: 0.463000; val_acc: 0.298000
(Epoch 6 / 10) train acc: 0.495000; val_acc: 0.293000
(Epoch 7 / 10) train acc: 0.529000; val_acc: 0.311000
(Epoch 8 / 10) train acc: 0.565000; val_acc: 0.308000
(Epoch 9 / 10) train acc: 0.661000; val_acc: 0.308000
(Epoch 10 / 10) train acc: 0.661000; val_acc: 0.313000
```

In [41]:
```
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()
```



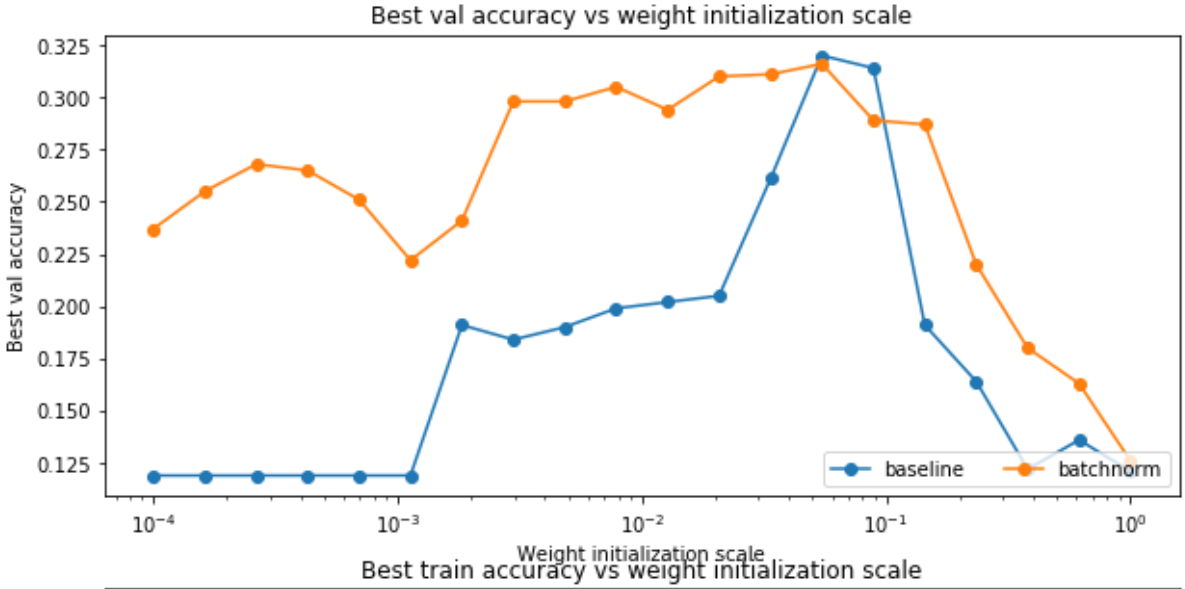
Batchnorm and initialization

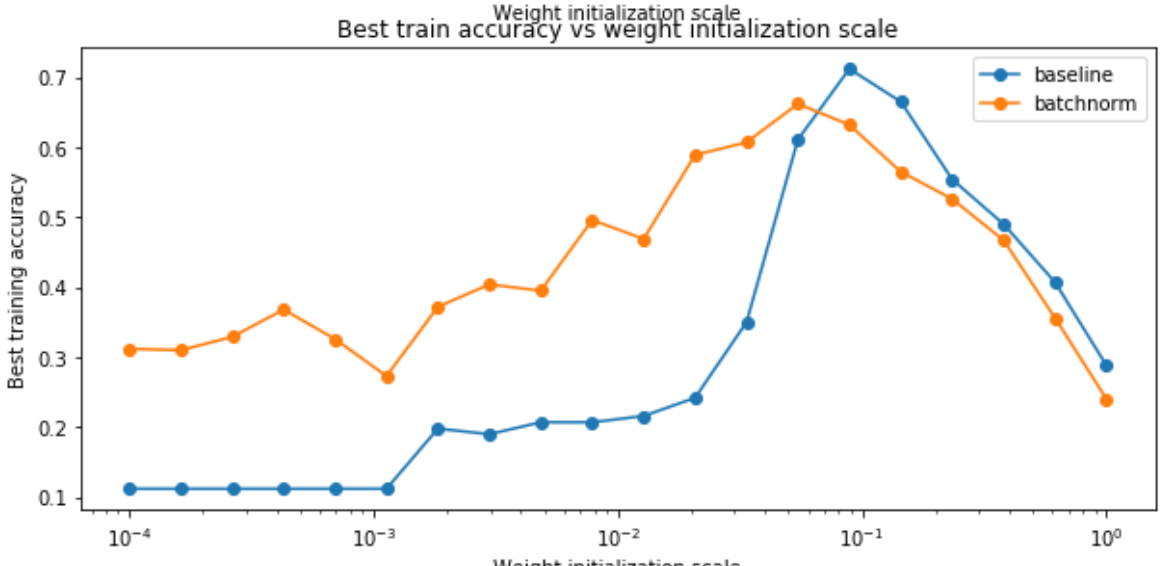
The following cells run an experiment where for a deep network, the initialization is varied. We do training for when batchnorm layers are and are not included.

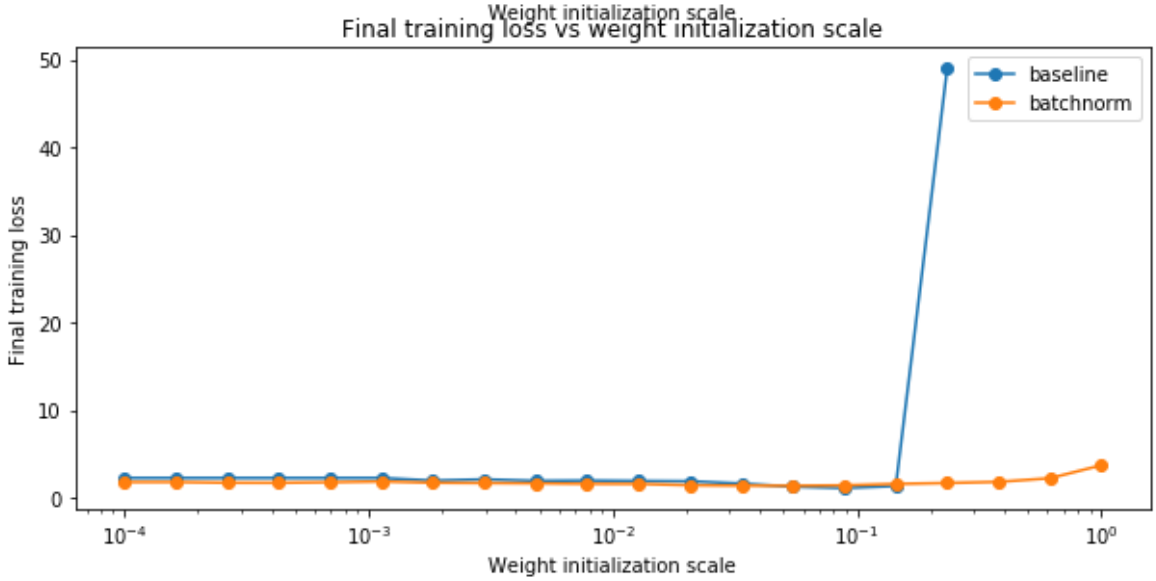
In [43]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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_scales
)))
  bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, u
se_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=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 [44]:
```
# 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='batchnorm')
plt.legend()

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

plt.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:

Batch Normalization tries to avert a lot of issues related to bad inititalization by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. This is possible because normalization is a simple differentiable operation.

Batch normalization can be interpreted as doing preprocessing at every layer of the network. Networks that use Batch Normalization are significantly more robust to bad initialization as it can be seen from the above plots, the network which uses batch norm has a better final training accuracy as compared to a baseline network that does not use batch norm.

The trainig loss for baseline network is much higher than that of the batch norm network thus confirming our proposition that performance of neural networks with batch norm is much better when initialization is random.

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 [17]:
```
## 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_numerica
l_gradient_array
from cs231n.solver import Solver

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

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

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

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

In [18]:
```
# 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 [20]:
```
x = np.random.randn(500, 500) + 10

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

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

```
('Running tests with p = ', 0.3)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 23.311412482547)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.30092)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.6)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 6.652111061656034)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.600916)
('Fraction of test-time output set to zero: ', 0.0)
('Running tests with p = ', 0.75)
('Mean of input: ', 10.001928680028106)
('Mean of train-time output: ', 3.3289192472660556)
('Mean of test-time output: ', 10.001928680028106)
('Fraction of train-time output set to zero: ', 0.75024)
('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 [21]:
```
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, dr
opout_param)[0], x, dout)

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

```
('dx relative error: ', 5.445607071257824e-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 [24]:
```
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=Fa
lse, h=1e-5)
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads
[name])))
  print('\n')
```

```
('Running check with dropout = ', 0)
('Initial loss: ', 2.298949958544263)
W1 relative error: 1.87119719924e-06
W2 relative error: 3.75955387515e-07
W3 relative error: 4.41558101425e-08
b1 relative error: 2.73991000184e-08
b2 relative error: 3.46941444851e-09
b3 relative error: 1.38519530797e-10


('Running check with dropout = ', 0.25)
('Initial loss: ', 2.2597611820017107)
W1 relative error: 5.90756092267e-09
W2 relative error: 1.30607120549e-07
W3 relative error: 1.2081297763e-08
b1 relative error: 5.47914274408e-10
b2 relative error: 1.23932214532e-07
b3 relative error: 5.96589419451e-11


('Running check with dropout = ', 0.5)
('Initial loss: ', 2.298135001122367)
W1 relative error: 2.19459352143e-08
W2 relative error: 1.8332201706e-08
W3 relative error: 2.0893611084e-08
b1 relative error: 6.06139640116e-10
b2 relative error: 1.63235192309e-09
b3 relative error: 1.2861046903e-10
```

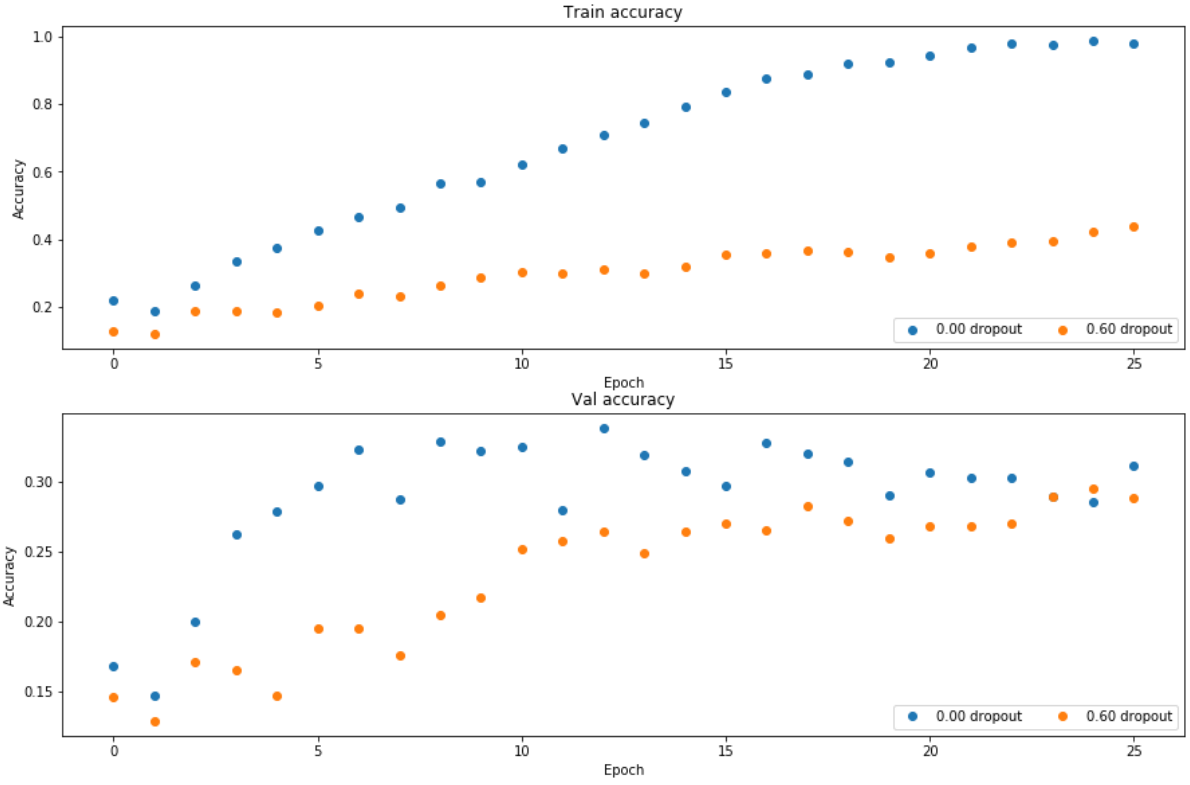
Dropout as a regularizer

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 [25]:
```
# 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: 2.301219
(Epoch 0 / 25) train acc: 0.132000; val_acc: 0.146000
(Epoch 1 / 25) train acc: 0.122000; val_acc: 0.129000
(Epoch 2 / 25) train acc: 0.190000; val_acc: 0.171000
(Epoch 3 / 25) train acc: 0.190000; val_acc: 0.165000
(Epoch 4 / 25) train acc: 0.186000; val_acc: 0.147000
(Epoch 5 / 25) train acc: 0.206000; val_acc: 0.195000
(Epoch 6 / 25) train acc: 0.240000; val_acc: 0.195000
(Epoch 7 / 25) train acc: 0.234000; val_acc: 0.176000
(Epoch 8 / 25) train acc: 0.264000; val_acc: 0.205000
(Epoch 9 / 25) train acc: 0.288000; val_acc: 0.217000
(Epoch 10 / 25) train acc: 0.306000; val_acc: 0.252000
(Epoch 11 / 25) train acc: 0.302000; val_acc: 0.257000
(Epoch 12 / 25) train acc: 0.312000; val_acc: 0.264000
(Epoch 13 / 25) train acc: 0.300000; val_acc: 0.249000
(Epoch 14 / 25) train acc: 0.322000; val_acc: 0.264000
(Epoch 15 / 25) train acc: 0.358000; val_acc: 0.270000
(Epoch 16 / 25) train acc: 0.360000; val_acc: 0.265000
(Epoch 17 / 25) train acc: 0.368000; val_acc: 0.282000
(Epoch 18 / 25) train acc: 0.364000; val_acc: 0.272000
(Epoch 19 / 25) train acc: 0.350000; val_acc: 0.259000
(Epoch 20 / 25) train acc: 0.360000; val_acc: 0.268000
(Iteration 101 / 125) loss: 1.839906
(Epoch 21 / 25) train acc: 0.380000; val_acc: 0.268000
(Epoch 22 / 25) train acc: 0.392000; val_acc: 0.270000
(Epoch 23 / 25) train acc: 0.396000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.424000; val_acc: 0.295000
(Epoch 25 / 25) train acc: 0.440000; val_acc: 0.288000
```

In [26]:
```
# Plot train and validation accuracies of the two models

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

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

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

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



Question

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

Answer:

Without dropout, we can see that trianing accuracy ~97% but validation accuracy is 31%. This shows that the model is overfitting. After dropout, training accuracy and Validation accurac ~ 40% and 30%. The training accuracy is comparable to validation accuracy. Therefore, the model is not overfitting which proves that dropout actually is sort of a regularization.

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 [56]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=50, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 24500) loss: 10.367319
(Epoch 0 / 50) train acc: 0.112000; val_acc: 0.132000
(Epoch 1 / 50) train acc: 0.453000; val_acc: 0.412000
(Iteration 501 / 24500) loss: 3.678347
(Epoch 2 / 50) train acc: 0.449000; val_acc: 0.453000
(Iteration 1001 / 24500) loss: 2.483102
(Epoch 3 / 50) train acc: 0.476000; val_acc: 0.479000
(Iteration 1501 / 24500) loss: 2.405102
(Epoch 4 / 50) train acc: 0.500000; val_acc: 0.472000
(Iteration 2001 / 24500) loss: 1.957018
(Epoch 5 / 50) train acc: 0.475000; val_acc: 0.484000
(Iteration 2501 / 24500) loss: 1.784672
(Epoch 6 / 50) train acc: 0.500000; val_acc: 0.480000
(Iteration 3001 / 24500) loss: 1.740386
(Epoch 7 / 50) train acc: 0.517000; val_acc: 0.502000
(Iteration 3501 / 24500) loss: 1.752235
(Epoch 8 / 50) train acc: 0.545000; val_acc: 0.509000
(Iteration 4001 / 24500) loss: 1.818646
(Epoch 9 / 50) train acc: 0.527000; val_acc: 0.504000
(Iteration 4501 / 24500) loss: 1.871535
(Epoch 10 / 50) train acc: 0.562000; val_acc: 0.524000
(Iteration 5001 / 24500) loss: 1.663305
(Epoch 11 / 50) train acc: 0.567000; val_acc: 0.533000
(Iteration 5501 / 24500) loss: 1.637834
(Epoch 12 / 50) train acc: 0.591000; val_acc: 0.533000
(Iteration 6001 / 24500) loss: 1.482642
(Epoch 13 / 50) train acc: 0.621000; val_acc: 0.562000
(Iteration 6501 / 24500) loss: 1.545891
(Epoch 14 / 50) train acc: 0.584000; val_acc: 0.575000
(Iteration 7001 / 24500) loss: 1.558609
(Epoch 15 / 50) train acc: 0.650000; val_acc: 0.561000
(Iteration 7501 / 24500) loss: 1.457756
(Epoch 16 / 50) train acc: 0.651000; val_acc: 0.572000
(Iteration 8001 / 24500) loss: 1.329249
(Epoch 17 / 50) train acc: 0.642000; val_acc: 0.577000
(Iteration 8501 / 24500) loss: 1.236984
(Epoch 18 / 50) train acc: 0.682000; val_acc: 0.564000
(Iteration 9001 / 24500) loss: 1.492818
(Epoch 19 / 50) train acc: 0.683000; val_acc: 0.579000
(Iteration 9501 / 24500) loss: 1.138019
(Epoch 20 / 50) train acc: 0.700000; val_acc: 0.570000
(Iteration 10001 / 24500) loss: 1.370272
(Epoch 21 / 50) train acc: 0.681000; val_acc: 0.567000
(Iteration 10501 / 24500) loss: 1.246665
(Epoch 22 / 50) train acc: 0.688000; val_acc: 0.580000
(Iteration 11001 / 24500) loss: 1.193318
(Epoch 23 / 50) train acc: 0.693000; val_acc: 0.580000
(Iteration 11501 / 24500) loss: 1.133695
(Epoch 24 / 50) train acc: 0.703000; val_acc: 0.587000
(Iteration 12001 / 24500) loss: 1.189112
(Epoch 25 / 50) train acc: 0.707000; val_acc: 0.584000
(Iteration 12501 / 24500) loss: 1.184437
(Epoch 26 / 50) train acc: 0.731000; val_acc: 0.585000
(Iteration 13001 / 24500) loss: 1.185885
(Epoch 27 / 50) train acc: 0.754000; val_acc: 0.582000
(Iteration 13501 / 24500) loss: 1.130842
(Epoch 28 / 50) train acc: 0.768000; val_acc: 0.586000
```

```
(Iteration 14001 / 24500) loss: 0.868604
(Epoch 29 / 50) train acc: 0.773000; val_acc: 0.592000
(Iteration 14501 / 24500) loss: 0.922187
(Epoch 30 / 50) train acc: 0.744000; val_acc: 0.584000
(Iteration 15001 / 24500) loss: 1.206035
(Epoch 31 / 50) train acc: 0.778000; val_acc: 0.592000
(Iteration 15501 / 24500) loss: 1.383444
(Epoch 32 / 50) train acc: 0.743000; val_acc: 0.592000
(Iteration 16001 / 24500) loss: 1.157079
(Epoch 33 / 50) train acc: 0.796000; val_acc: 0.587000
(Iteration 16501 / 24500) loss: 1.108257
(Epoch 34 / 50) train acc: 0.777000; val_acc: 0.602000
(Iteration 17001 / 24500) loss: 0.997282
(Epoch 35 / 50) train acc: 0.793000; val_acc: 0.597000
(Iteration 17501 / 24500) loss: 0.914432
(Epoch 36 / 50) train acc: 0.798000; val_acc: 0.596000
(Iteration 18001 / 24500) loss: 0.956723
(Epoch 37 / 50) train acc: 0.811000; val_acc: 0.583000
(Iteration 18501 / 24500) loss: 1.203487
(Epoch 38 / 50) train acc: 0.792000; val_acc: 0.599000
(Iteration 19001 / 24500) loss: 1.036741
(Epoch 39 / 50) train acc: 0.793000; val_acc: 0.600000
(Iteration 19501 / 24500) loss: 0.952084
(Epoch 40 / 50) train acc: 0.828000; val_acc: 0.595000
(Iteration 20001 / 24500) loss: 0.871738
(Epoch 41 / 50) train acc: 0.818000; val_acc: 0.595000
(Iteration 20501 / 24500) loss: 0.961417
(Epoch 42 / 50) train acc: 0.821000; val_acc: 0.593000
(Iteration 21001 / 24500) loss: 0.945501
(Epoch 43 / 50) train acc: 0.816000; val_acc: 0.594000
(Iteration 21501 / 24500) loss: 0.784905
(Epoch 44 / 50) train acc: 0.845000; val_acc: 0.593000
(Iteration 22001 / 24500) loss: 0.953447
(Epoch 45 / 50) train acc: 0.823000; val_acc: 0.598000
(Iteration 22501 / 24500) loss: 0.893920
(Epoch 46 / 50) train acc: 0.823000; val_acc: 0.602000
(Iteration 23001 / 24500) loss: 0.797951
(Epoch 47 / 50) train acc: 0.835000; val_acc: 0.598000
(Iteration 23501 / 24500) loss: 0.949376
(Epoch 48 / 50) train acc: 0.797000; val_acc: 0.596000
(Iteration 24001 / 24500) loss: 0.955032
(Epoch 49 / 50) train acc: 0.836000; val_acc: 0.593000
(Epoch 50 / 50) train acc: 0.833000; val_acc: 0.594000
```

In [62]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=34, batch_size=128,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 12988) loss: 10.345334
(Epoch 0 / 34) train acc: 0.116000; val_acc: 0.107000
(Epoch 1 / 34) train acc: 0.481000; val_acc: 0.433000
(Iteration 501 / 12988) loss: 3.388941
(Epoch 2 / 34) train acc: 0.470000; val_acc: 0.475000
(Iteration 1001 / 12988) loss: 2.470986
(Epoch 3 / 34) train acc: 0.492000; val_acc: 0.495000
(Iteration 1501 / 12988) loss: 1.965256
(Epoch 4 / 34) train acc: 0.494000; val_acc: 0.500000
(Epoch 5 / 34) train acc: 0.495000; val_acc: 0.480000
(Iteration 2001 / 12988) loss: 1.982240
(Epoch 6 / 34) train acc: 0.540000; val_acc: 0.498000
(Iteration 2501 / 12988) loss: 1.784661
(Epoch 7 / 34) train acc: 0.538000; val_acc: 0.509000
(Iteration 3001 / 12988) loss: 1.667780
(Epoch 8 / 34) train acc: 0.559000; val_acc: 0.510000
(Epoch 9 / 34) train acc: 0.552000; val_acc: 0.537000
(Iteration 3501 / 12988) loss: 1.809124
(Epoch 10 / 34) train acc: 0.569000; val_acc: 0.546000
(Iteration 4001 / 12988) loss: 1.487143
(Epoch 11 / 34) train acc: 0.591000; val_acc: 0.531000
(Iteration 4501 / 12988) loss: 1.475054
(Epoch 12 / 34) train acc: 0.569000; val_acc: 0.547000
(Epoch 13 / 34) train acc: 0.622000; val_acc: 0.556000
(Iteration 5001 / 12988) loss: 1.589821
(Epoch 14 / 34) train acc: 0.621000; val_acc: 0.559000
(Iteration 5501 / 12988) loss: 1.355803
(Epoch 15 / 34) train acc: 0.611000; val_acc: 0.562000
(Iteration 6001 / 12988) loss: 1.367220
(Epoch 16 / 34) train acc: 0.625000; val_acc: 0.569000
(Epoch 17 / 34) train acc: 0.677000; val_acc: 0.574000
(Iteration 6501 / 12988) loss: 1.300967
(Epoch 18 / 34) train acc: 0.659000; val_acc: 0.588000
(Iteration 7001 / 12988) loss: 1.230683
(Epoch 19 / 34) train acc: 0.684000; val_acc: 0.574000
(Iteration 7501 / 12988) loss: 1.341910
(Epoch 20 / 34) train acc: 0.701000; val_acc: 0.574000
(Iteration 8001 / 12988) loss: 1.442092
(Epoch 21 / 34) train acc: 0.710000; val_acc: 0.575000
(Epoch 22 / 34) train acc: 0.728000; val_acc: 0.585000
(Iteration 8501 / 12988) loss: 1.237475
(Epoch 23 / 34) train acc: 0.727000; val_acc: 0.592000
(Iteration 9001 / 12988) loss: 1.103268
(Epoch 24 / 34) train acc: 0.698000; val_acc: 0.585000
(Iteration 9501 / 12988) loss: 1.191798
(Epoch 25 / 34) train acc: 0.767000; val_acc: 0.598000
(Epoch 26 / 34) train acc: 0.736000; val_acc: 0.595000
(Iteration 10001 / 12988) loss: 1.130865
(Epoch 27 / 34) train acc: 0.750000; val_acc: 0.594000
(Iteration 10501 / 12988) loss: 1.070354
(Epoch 28 / 34) train acc: 0.747000; val_acc: 0.594000
(Iteration 11001 / 12988) loss: 0.901335
(Epoch 29 / 34) train acc: 0.779000; val_acc: 0.593000
(Epoch 30 / 34) train acc: 0.762000; val_acc: 0.589000
(Iteration 11501 / 12988) loss: 1.099542
(Epoch 31 / 34) train acc: 0.784000; val_acc: 0.607000
(Iteration 12001 / 12988) loss: 0.901268
```

```
(Epoch 32 / 34) train acc: 0.769000; val_acc: 0.603000
(Iteration 12501 / 12988) loss: 0.901700
(Epoch 33 / 34) train acc: 0.767000; val_acc: 0.582000
(Epoch 34 / 34) train acc: 0.815000; val_acc: 0.599000
```

In [64]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 60% validation accuracy
#   on CIFAR-10.
# ================================================================ #
learning_rate = 3.1e-4
weight_scale = 2.5e-2 #1e-5
model = FullyConnectedNet([600, 500, 400, 300, 200, 100],
                weight_scale=weight_scale, dtype=np.float64, dropout=0.2
5, use_batchnorm=True, reg=1e-2)
solver = Solver(model, data,
                print_every=500, num_epochs=40, batch_size=128,
                update_rule='adam',
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=0.9
         )

solver.train()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 15280) loss: 10.338546
(Epoch 0 / 40) train acc: 0.097000; val_acc: 0.093000
(Epoch 1 / 40) train acc: 0.434000; val_acc: 0.429000
(Iteration 501 / 15280) loss: 3.395782
(Epoch 2 / 40) train acc: 0.440000; val_acc: 0.459000
(Iteration 1001 / 15280) loss: 2.147055
(Epoch 3 / 40) train acc: 0.494000; val_acc: 0.468000
(Iteration 1501 / 15280) loss: 2.026479
(Epoch 4 / 40) train acc: 0.498000; val_acc: 0.482000
(Epoch 5 / 40) train acc: 0.494000; val_acc: 0.507000
(Iteration 2001 / 15280) loss: 1.818237
(Epoch 6 / 40) train acc: 0.536000; val_acc: 0.505000
(Iteration 2501 / 15280) loss: 1.817284
(Epoch 7 / 40) train acc: 0.538000; val_acc: 0.528000
(Iteration 3001 / 15280) loss: 1.670779
(Epoch 8 / 40) train acc: 0.550000; val_acc: 0.534000
(Epoch 9 / 40) train acc: 0.582000; val_acc: 0.538000
(Iteration 3501 / 15280) loss: 1.703494
(Epoch 10 / 40) train acc: 0.538000; val_acc: 0.535000
(Iteration 4001 / 15280) loss: 1.724836
(Epoch 11 / 40) train acc: 0.587000; val_acc: 0.547000
(Iteration 4501 / 15280) loss: 1.524487
(Epoch 12 / 40) train acc: 0.567000; val_acc: 0.548000
(Epoch 13 / 40) train acc: 0.607000; val_acc: 0.558000
(Iteration 5001 / 15280) loss: 1.630233
(Epoch 14 / 40) train acc: 0.611000; val_acc: 0.545000
(Iteration 5501 / 15280) loss: 1.350028
(Epoch 15 / 40) train acc: 0.643000; val_acc: 0.553000
(Iteration 6001 / 15280) loss: 1.298213
(Epoch 16 / 40) train acc: 0.678000; val_acc: 0.552000
(Epoch 17 / 40) train acc: 0.671000; val_acc: 0.583000
(Iteration 6501 / 15280) loss: 1.299723
(Epoch 18 / 40) train acc: 0.672000; val_acc: 0.575000
(Iteration 7001 / 15280) loss: 1.384886
(Epoch 19 / 40) train acc: 0.646000; val_acc: 0.585000
(Iteration 7501 / 15280) loss: 1.313679
(Epoch 20 / 40) train acc: 0.677000; val_acc: 0.577000
(Iteration 8001 / 15280) loss: 1.364944
(Epoch 21 / 40) train acc: 0.688000; val_acc: 0.580000
(Epoch 22 / 40) train acc: 0.732000; val_acc: 0.586000
(Iteration 8501 / 15280) loss: 1.311282
(Epoch 23 / 40) train acc: 0.725000; val_acc: 0.589000
(Iteration 9001 / 15280) loss: 1.216050
(Epoch 24 / 40) train acc: 0.724000; val_acc: 0.577000
(Iteration 9501 / 15280) loss: 1.147909
(Epoch 25 / 40) train acc: 0.756000; val_acc: 0.598000
(Epoch 26 / 40) train acc: 0.718000; val_acc: 0.608000
(Iteration 10001 / 15280) loss: 1.139750
(Epoch 27 / 40) train acc: 0.744000; val_acc: 0.583000
(Iteration 10501 / 15280) loss: 0.973199
(Epoch 28 / 40) train acc: 0.735000; val_acc: 0.590000
(Iteration 11001 / 15280) loss: 1.104056
(Epoch 29 / 40) train acc: 0.763000; val_acc: 0.591000
(Epoch 30 / 40) train acc: 0.791000; val_acc: 0.582000
(Iteration 11501 / 15280) loss: 0.944136
(Epoch 31 / 40) train acc: 0.771000; val_acc: 0.588000
(Iteration 12001 / 15280) loss: 1.012222
```

```
(Epoch 32 / 40) train acc: 0.798000; val_acc: 0.585000
(Iteration 12501 / 15280) loss: 1.036070
(Epoch 33 / 40) train acc: 0.787000; val_acc: 0.591000
(Epoch 34 / 40) train acc: 0.788000; val_acc: 0.585000
(Iteration 13001 / 15280) loss: 0.927312
(Epoch 35 / 40) train acc: 0.784000; val_acc: 0.598000
(Iteration 13501 / 15280) loss: 1.047863
(Epoch 36 / 40) train acc: 0.794000; val_acc: 0.594000
(Iteration 14001 / 15280) loss: 1.043726
(Epoch 37 / 40) train acc: 0.800000; val_acc: 0.590000
(Iteration 14501 / 15280) loss: 0.920529
(Epoch 38 / 40) train acc: 0.810000; val_acc: 0.588000
(Epoch 39 / 40) train acc: 0.826000; val_acc: 0.588000
(Iteration 15001 / 15280) loss: 0.863095
(Epoch 40 / 40) train acc: 0.824000; val_acc: 0.594000
```

In [ ]:
```
Validation Accuracy of ~60% achived in earlie Epochs.
```