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.

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

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 [3]:
```
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: 9.769849468192957e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 1.7080572793816912e-10
dw error: 3.635605517502198e-11
db error: 8.865763799774247e-11


If relu_forward function is working, difference should be around 1e-8:
difference: 4.999999798022158e-08


If relu_forward function is working, error should be less than 1e-9:
dx error: 3.2756227385966328e-12


If affine_relu_forward and affine_relu_backward are working, error should
be less than 1e-9::
dx error: 3.375302606376472e-11
dw error: 1.718953092920777e-10
db error: 3.2755304933243622e-12


Running check with reg = 0
Initial loss: 2.3036302335690655
W1 relative error: 1.874059469406586e-07
W2 relative error: 2.7718383193017137e-07
W3 relative error: 7.487491764797539e-07
b1 relative error: 1.0840298013034233e-08
b2 relative error: 2.192153290300306e-09
b3 relative error: 8.084625476723435e-11
Running check with reg = 3.14
Initial loss: 6.929910430205594
W1 relative error: 1.3875332034181178e-07
W2 relative error: 7.526880271955046e-08
W3 relative error: 4.1706873173251974e-08
b1 relative error: 9.70713051820796e-08
b2 relative error: 4.06193540421361e-08
b3 relative error: 1.5300450731016278e-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 [4]:
```
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: 8.882347033505819e-09
velocity error: 4.269287743278663e-09
```

SGD + Nesterov momentum

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

In [5]:
```
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'])))
```

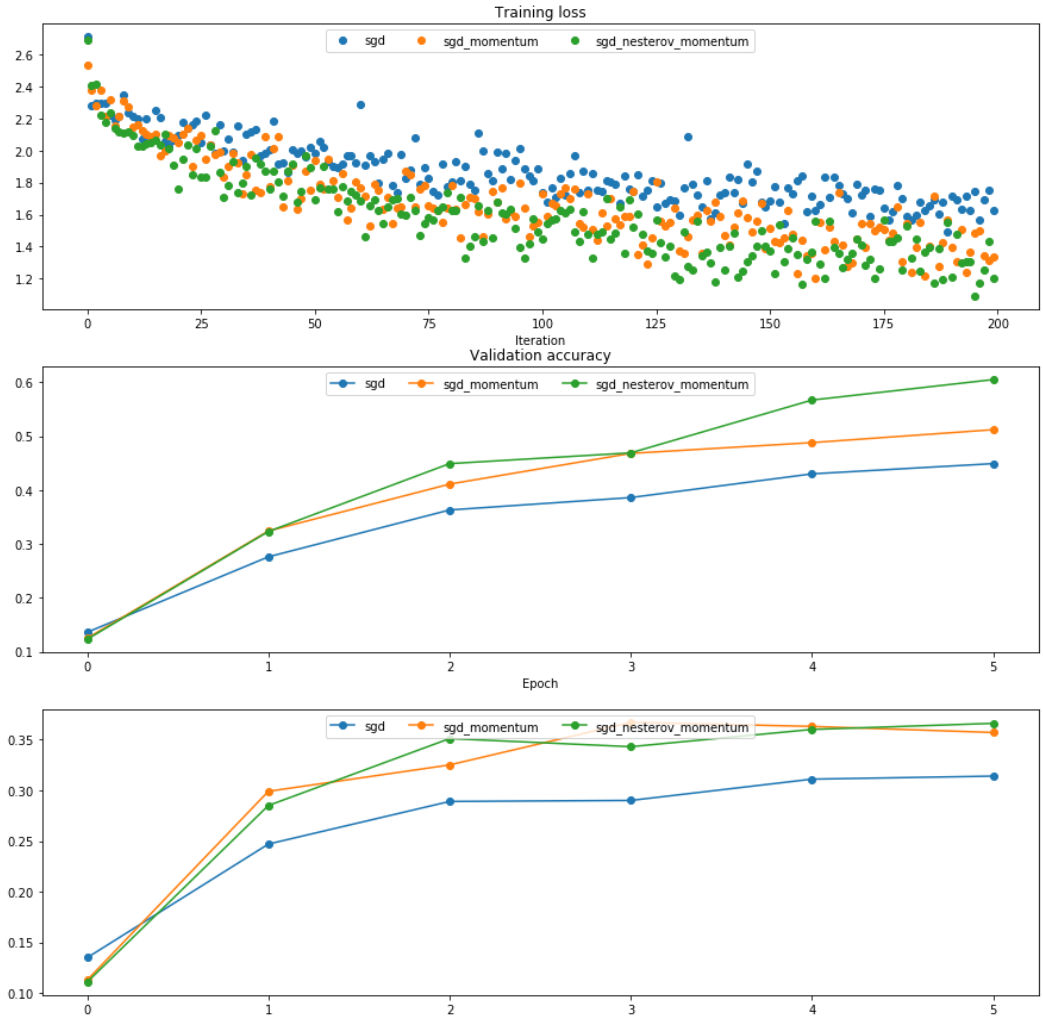
```
next_w error: 1.0875186845081027e-08
velocity error: 4.269287743278663e-09
```

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 [6]:
```
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

fig, axes = plt.subplots(3, 1)

ax = axes[0]
ax.set_title('Training loss')
ax.set_xlabel('Iteration')

ax = axes[1]
ax.set_title('Training accuracy')
ax.set_xlabel('Epoch')

ax = axes[1]
ax.set_title('Validation accuracy')
ax.set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    ax = axes[0]
    ax.plot(solver.loss_history, 'o', label=update_rule)

    ax = axes[1]
    ax.plot(solver.train_acc_history, '-o', label=update_rule)

    ax = axes[2]
    ax.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
    ax = axes[i - 1]
    ax.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 [7]:
```
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: 9.524687511038133e-08
cache error: 2.6477955807156126e-09
```

Adaptive moments

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

In [8]:
```
# 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'])))
```

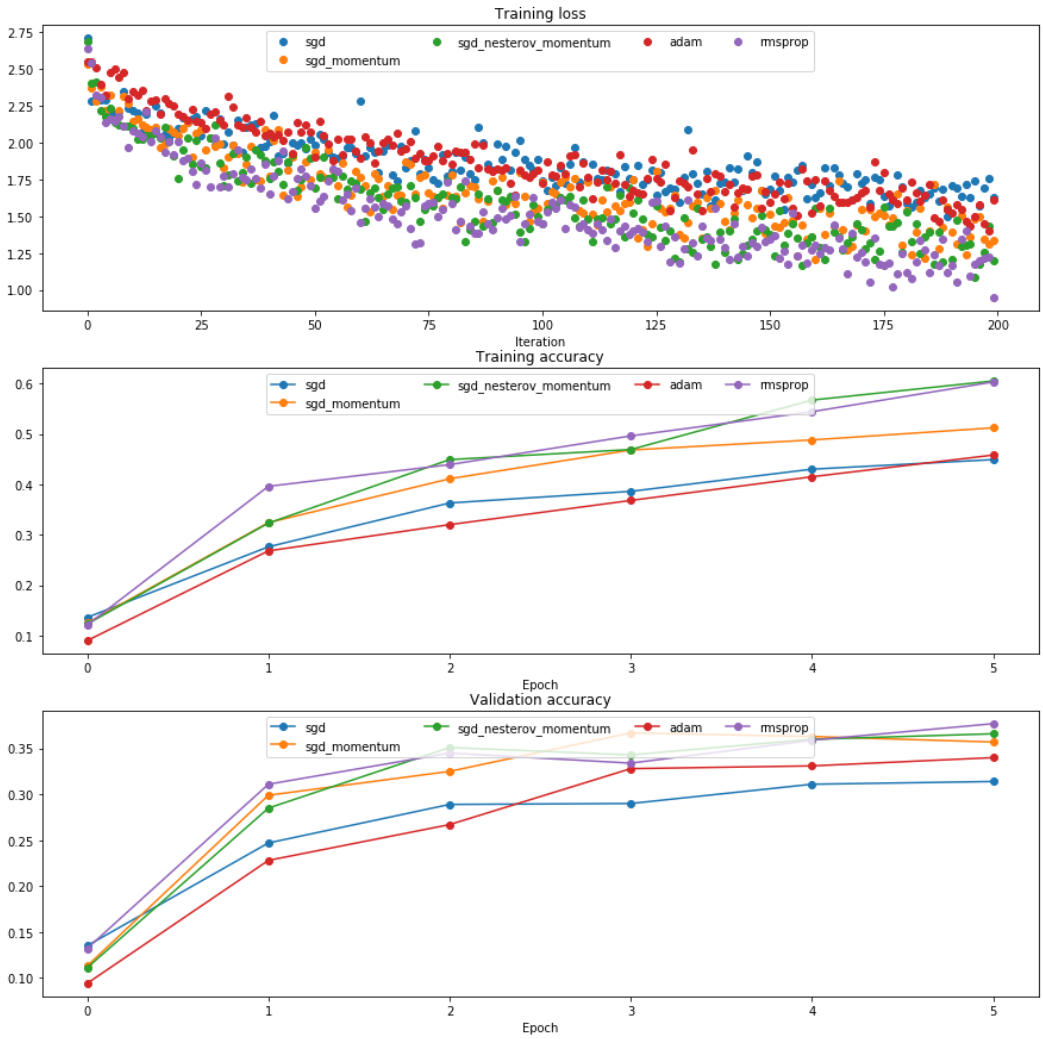
```
next_w error: 8.699924736323826e-05
a error: 0.024836929252383348
v error: 4.214963193114416e-09
```

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 [9]:
```
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

fig, axes = plt.subplots(3, 1)

ax = axes[0]
ax.set_title('Training loss')
ax.set_xlabel('Iteration')

ax = axes[1]
ax.set_title('Training accuracy')
ax.set_xlabel('Epoch')

ax = axes[2]
ax.set_title('Validation accuracy')
ax.set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    ax = axes[0]
    ax.plot(solver.loss_history, 'o', label=update_rule)

    ax = axes[1]
    ax.plot(solver.train_acc_history, '-o', label=update_rule)

    ax = axes[2]
    ax.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
    ax = axes[i - 1]
    ax.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 55+% on CIFAR-10.

In [10]:
```
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: 2.262451
(Epoch 0 / 10) train acc: 0.140000; val_acc: 0.155000
(Iteration 51 / 4900) loss: 1.777307
(Iteration 101 / 4900) loss: 1.659956
(Iteration 151 / 4900) loss: 1.613884
(Iteration 201 / 4900) loss: 1.600903
(Iteration 251 / 4900) loss: 1.499103
(Iteration 301 / 4900) loss: 1.413548
(Iteration 351 / 4900) loss: 1.377646
(Iteration 401 / 4900) loss: 1.334849
(Iteration 451 / 4900) loss: 1.427387
(Epoch 1 / 10) train acc: 0.524000; val_acc: 0.499000
(Iteration 501 / 4900) loss: 1.409675
(Iteration 551 / 4900) loss: 1.390866
(Iteration 601 / 4900) loss: 1.336394
(Iteration 651 / 4900) loss: 1.517225
(Iteration 701 / 4900) loss: 1.281355
(Iteration 751 / 4900) loss: 1.217513
(Iteration 801 / 4900) loss: 1.404522
(Iteration 851 / 4900) loss: 1.244994
(Iteration 901 / 4900) loss: 1.250716
(Iteration 951 / 4900) loss: 1.307781
(Epoch 2 / 10) train acc: 0.552000; val_acc: 0.526000
(Iteration 1001 / 4900) loss: 1.252200
(Iteration 1051 / 4900) loss: 1.122882
(Iteration 1101 / 4900) loss: 0.991882
(Iteration 1151 / 4900) loss: 1.171788
(Iteration 1201 / 4900) loss: 1.150502
(Iteration 1251 / 4900) loss: 1.176916
(Iteration 1301 / 4900) loss: 0.997382
(Iteration 1351 / 4900) loss: 1.106426
(Iteration 1401 / 4900) loss: 1.003027
(Iteration 1451 / 4900) loss: 1.078969
(Epoch 3 / 10) train acc: 0.605000; val_acc: 0.544000
(Iteration 1501 / 4900) loss: 1.132291
(Iteration 1551 / 4900) loss: 1.048286
(Iteration 1601 / 4900) loss: 0.969781
(Iteration 1651 / 4900) loss: 1.172605
(Iteration 1701 / 4900) loss: 1.013117
(Iteration 1751 / 4900) loss: 0.985120
(Iteration 1801 / 4900) loss: 0.850634
(Iteration 1851 / 4900) loss: 1.064179
(Iteration 1901 / 4900) loss: 0.962034
(Iteration 1951 / 4900) loss: 0.887884
(Epoch 4 / 10) train acc: 0.684000; val_acc: 0.546000
(Iteration 2001 / 4900) loss: 0.969318
(Iteration 2051 / 4900) loss: 0.902149
(Iteration 2101 / 4900) loss: 0.910347
(Iteration 2151 / 4900) loss: 0.856774
(Iteration 2201 / 4900) loss: 0.937697
(Iteration 2251 / 4900) loss: 0.958938
(Iteration 2301 / 4900) loss: 0.839748
(Iteration 2351 / 4900) loss: 0.984624
(Iteration 2401 / 4900) loss: 1.084647
(Epoch 5 / 10) train acc: 0.689000; val_acc: 0.541000
(Iteration 2451 / 4900) loss: 1.053752
(Iteration 2501 / 4900) loss: 0.997177
(Iteration 2551 / 4900) loss: 0.750070
(Iteration 2601 / 4900) loss: 0.736606
(Iteration 2651 / 4900) loss: 0.881684
(Iteration 2701 / 4900) loss: 0.812084
(Iteration 2751 / 4900) loss: 0.743127
(Iteration 2801 / 4900) loss: 0.768724
(Iteration 2851 / 4900) loss: 0.742728
(Iteration 2901 / 4900) loss: 0.886776
(Epoch 6 / 10) train acc: 0.720000; val_acc: 0.553000
(Iteration 2951 / 4900) loss: 0.649285
(Iteration 3001 / 4900) loss: 0.861470
```

In [11]:
```
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: 0.571
Test set accuracy: 0.568
```

Batch Normalization

In this notebook, you will implement the batch normalization layers of a neural network to increase its performance. 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_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

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:  [ 13.35649799  24.61421955 -14.15107081]
  stds:  [35.56561098 35.15974973 31.91019606]
After batch normalization (gamma=1, beta=0)
  mean:  [-2.38697950e-16 -2.39808173e-16 -4.60742555e-17]
  std:  [1. 1. 1.]
After batch normalization (nontrivial gamma, beta)
  means:  [11. 12. 13.]
  stds:  [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:  [-0.05967139  0.00957257 -0.0259153 ]
  stds:  [1.0150653  0.99071727 1.00119239]
```

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:  1.0365309983627649e-08
dgamma error:  4.235503173805597e-12
dbeta error:  3.275576553527879e-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 [6]:
```
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.float
64,
                              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], verbos
e=False, h=1e-5)
        print('{} relative error: {}'.format(name, rel_error(grad_num, g
rads[name])))
    if reg == 0: print('\n')
```

```
Running check with reg =  0
Initial loss:  2.495181649437833
W1 relative error: 5.7133128371328395e-05
W2 relative error: 6.152107138686845e-06
W3 relative error: 3.860542911698204e-10
b1 relative error: 2.220446049250313e-08
b2 relative error: 5.551115123125783e-09
b3 relative error: 1.303595543107751e-10
beta1 relative error: 4.66520471937565e-08
beta2 relative error: 1.616541206944714e-09
gamma1 relative error: 3.0691595985183625e-08
gamma2 relative error: 2.3542593993916706e-09


Running check with reg =  3.14
Initial loss:  6.988149746148828
W1 relative error: 3.63450880030614e-07
W2 relative error: 1.4399569445718448e-08
W3 relative error: 6.072364329088387e-08
b1 relative error: 1.734723475976807e-10
b2 relative error: 2.7755575615628914e-09
b3 relative error: 1.4826365551616593e-10
beta1 relative error: 3.12891605875848e-06
beta2 relative error: 4.003822494515134e-08
gamma1 relative error: 4.038265200968759e-06
gamma2 relative error: 3.0722719270750266e-08
```

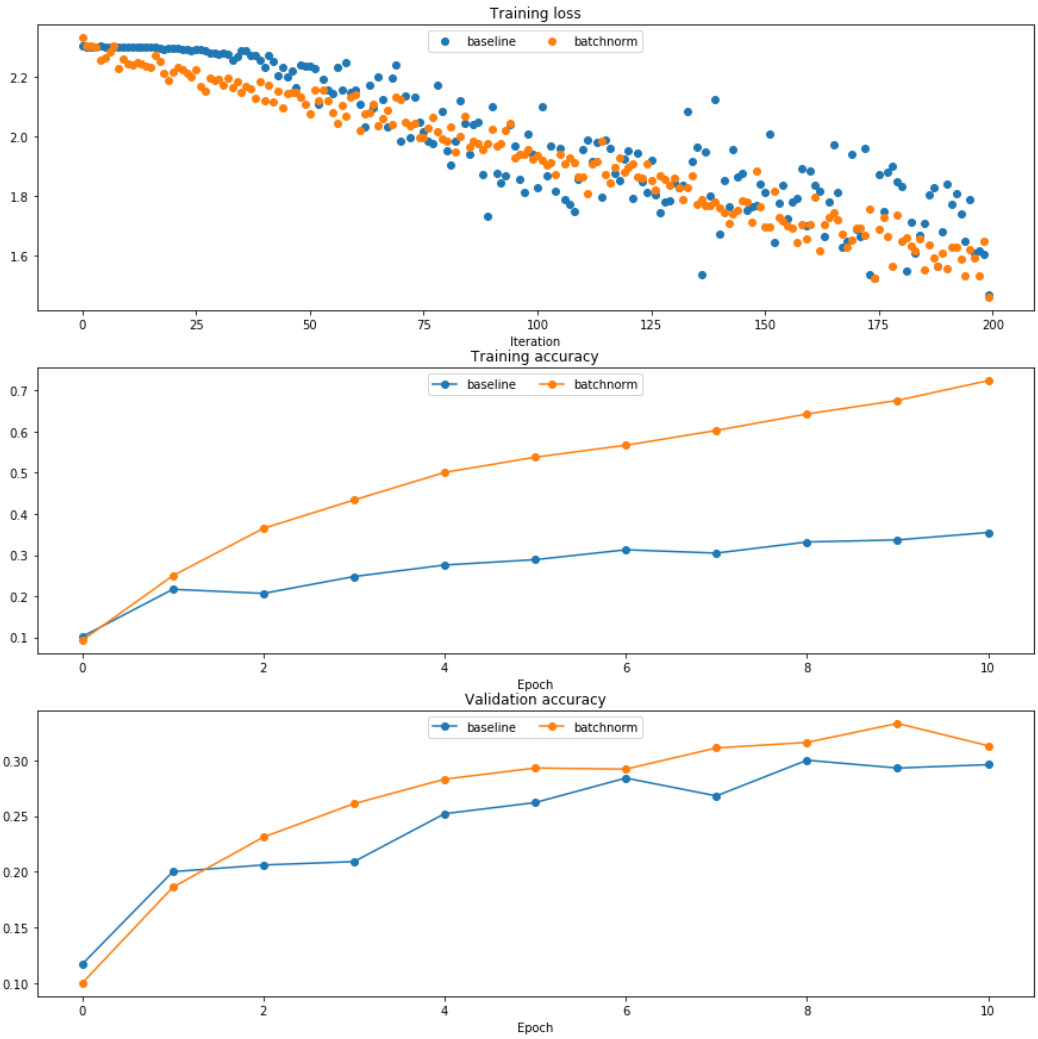
Training a deep fully connected network with batch normalization.

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

In [7]:
```
# 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: 2.330732
(Epoch 0 / 10) train acc: 0.093000; val_acc: 0.100000
(Epoch 1 / 10) train acc: 0.250000; val_acc: 0.186000
(Epoch 2 / 10) train acc: 0.365000; val_acc: 0.231000
(Epoch 3 / 10) train acc: 0.434000; val_acc: 0.261000
(Epoch 4 / 10) train acc: 0.501000; val_acc: 0.283000
(Epoch 5 / 10) train acc: 0.538000; val_acc: 0.293000
(Epoch 6 / 10) train acc: 0.567000; val_acc: 0.292000
(Epoch 7 / 10) train acc: 0.603000; val_acc: 0.311000
(Epoch 8 / 10) train acc: 0.643000; val_acc: 0.316000
(Epoch 9 / 10) train acc: 0.676000; val_acc: 0.333000
(Epoch 10 / 10) train acc: 0.724000; val_acc: 0.313000
(Iteration 1 / 200) loss: 2.303350
(Epoch 0 / 10) train acc: 0.102000; val_acc: 0.117000
(Epoch 1 / 10) train acc: 0.217000; val_acc: 0.200000
(Epoch 2 / 10) train acc: 0.207000; val_acc: 0.206000
(Epoch 3 / 10) train acc: 0.248000; val_acc: 0.209000
(Epoch 4 / 10) train acc: 0.276000; val_acc: 0.252000
(Epoch 5 / 10) train acc: 0.289000; val_acc: 0.262000
(Epoch 6 / 10) train acc: 0.313000; val_acc: 0.284000
(Epoch 7 / 10) train acc: 0.305000; val_acc: 0.268000
(Epoch 8 / 10) train acc: 0.332000; val_acc: 0.300000
(Epoch 9 / 10) train acc: 0.337000; val_acc: 0.293000
(Epoch 10 / 10) train acc: 0.355000; val_acc: 0.296000
```

In [8]:
```
fig, axes = plt.subplots(3, 1)

ax = axes[0]
ax.set_title('Training loss')
ax.set_xlabel('Iteration')

ax = axes[1]
ax.set_title('Training accuracy')
ax.set_xlabel('Epoch')

ax = axes[2]
ax.set_title('Validation accuracy')
ax.set_xlabel('Epoch')

ax = axes[0]
ax.plot(solver.loss_history, 'o', label='baseline')
ax.plot(bn_solver.loss_history, 'o', label='batchnorm')

ax = axes[1]
ax.plot(solver.train_acc_history, '-o', label='baseline')
ax.plot(bn_solver.train_acc_history, '-o', label='batchnorm')

ax = axes[2]
ax.plot(solver.val_acc_history, '-o', label='baseline')
ax.plot(bn_solver.val_acc_history, '-o', label='batchnorm')

for i in [1, 2, 3]:
    ax = axes[i - 1]
    ax.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.

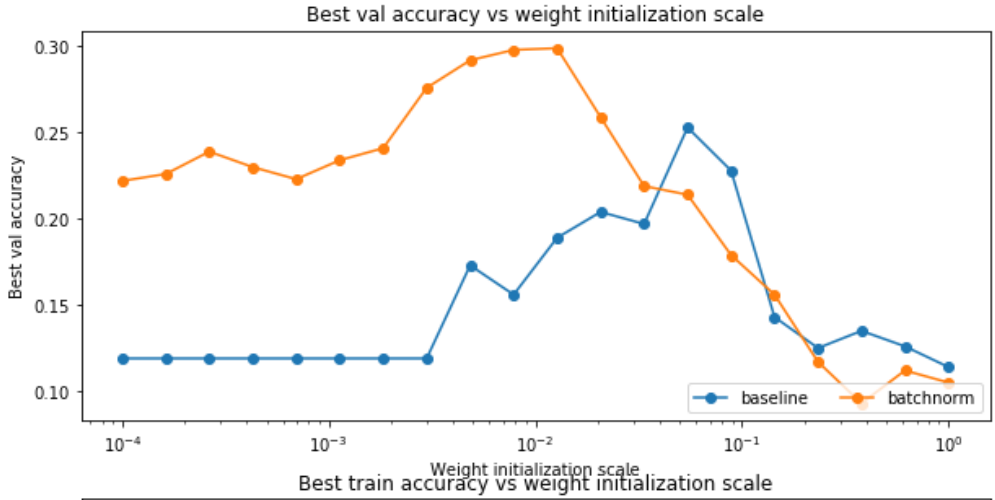
In [9]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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, us
e_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
```

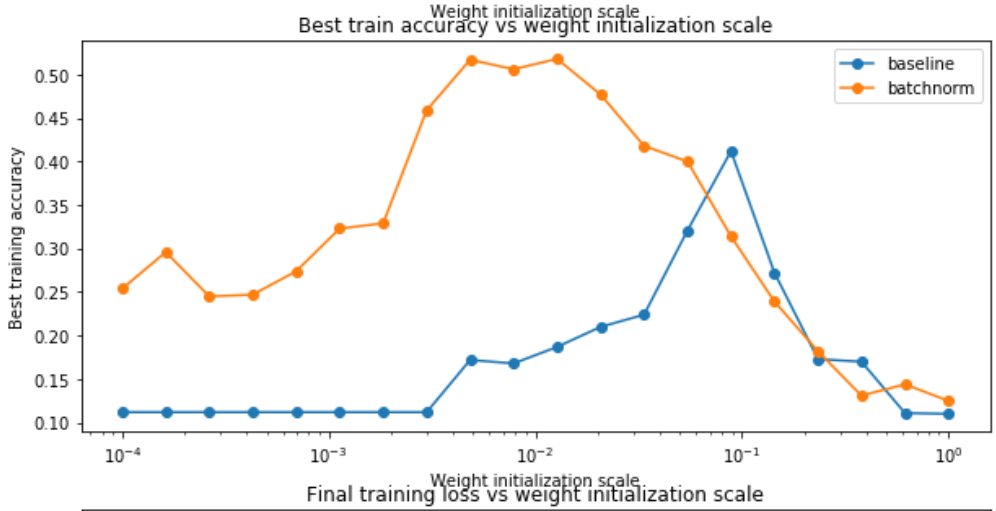
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 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
```

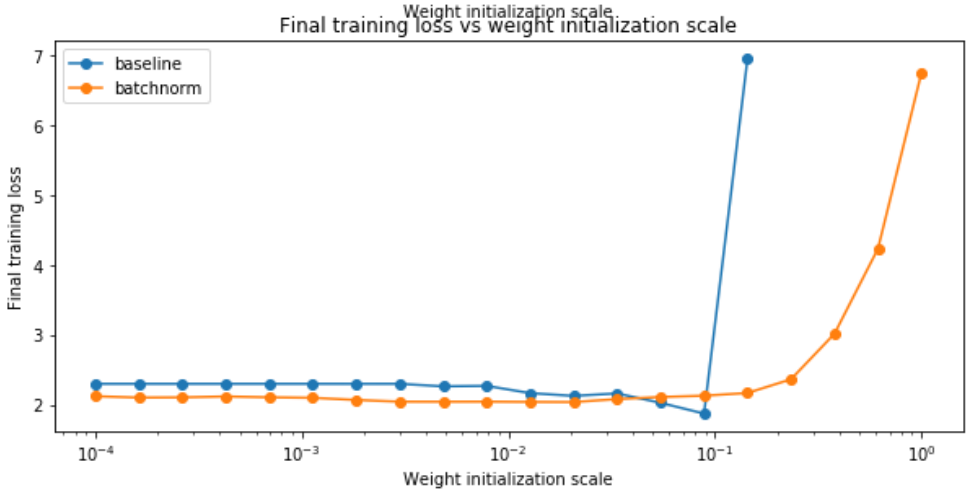
```
/home/dennis/Documents/PY_PROGRAM/UCLA_C247/HW4-code/HW4-code/nndl/layer
s.py:446: RuntimeWarning: divide by zero encountered in log
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
```

```
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

In [10]:
```
# 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[-10
0:]))

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:

On one hand, compared to the case without batch normalization, training with batch normalization allows us to train with a less bigger weight initialization scale, which is 2e-3, while the case without batch normalization is 5e-3. On the other hand, training with batch normalization tolerates greater weight initialization scale, meaning that training loss does not goes to ifinity with 1e-1 weight initialization scale. The main reason is that batch normalization acts as a regularization, which to some extent reduces the possibility for each layer of going to extreme values.

Dropout

In this notebook, you will implement dropout. Then we will ask you to train a network with batchnorm and dropout, and acheive over 55% 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_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_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

Dropout forward pass

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).mean
())
    print('Fraction of test-time output set to zero: ', (out_test == 0).
mean())
```

```
Running tests with p =  0.3
Mean of input:  9.997543371838061
Mean of train-time output:  9.99068001546249
Mean of test-time output:  9.997543371838061
Fraction of train-time output set to zero:  0.300588
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.997543371838061
Mean of train-time output:  9.951918706035858
Mean of test-time output:  9.997543371838061
Fraction of train-time output set to zero:  0.60184
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.997543371838061
Mean of train-time output:  10.016241307563895
Mean of test-time output:  9.997543371838061
Fraction of train-time output set to zero:  0.74962
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, dr
opout_param)[0], x, dout)

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

```
dx relative error:  1.8929088797253644e-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 [0.5, 0.75, 1.0]:
    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], verbos
e=False, h=1e-5)
        print('{} relative error: {}'.format(name, rel_error(grad_num, g
rads[name])))
    print('\n')
```

```
Running check with dropout =  0.5
Initial loss:  2.3130766680585553
W1 relative error: 3.521651327461513e-07
W2 relative error: 8.67050333013485e-08
W3 relative error: 1.7697091527236745e-08
b1 relative error: 1.3630944494006748e-09
b2 relative error: 3.369320802894315e-10
b3 relative error: 1.1749705566445823e-10


Running check with dropout =  0.75
Initial loss:  2.3021392488536803
W1 relative error: 8.48825148891291e-08
W2 relative error: 7.320957078668507e-10
W3 relative error: 3.1539905478679266e-09
b1 relative error: 4.991935528845047e-10
b2 relative error: 2.3792592457209184e-10
b3 relative error: 1.480081274701072e-10


Running check with dropout =  1.0
Initial loss:  2.3053332250963194
W1 relative error: 1.2744095365229032e-06
W2 relative error: 4.678743300473988e-07
W3 relative error: 4.331673892536035e-08
b1 relative error: 4.0853539035931665e-08
b2 relative error: 1.951342257912746e-09
b3 relative error: 9.387142701440351e-11
```

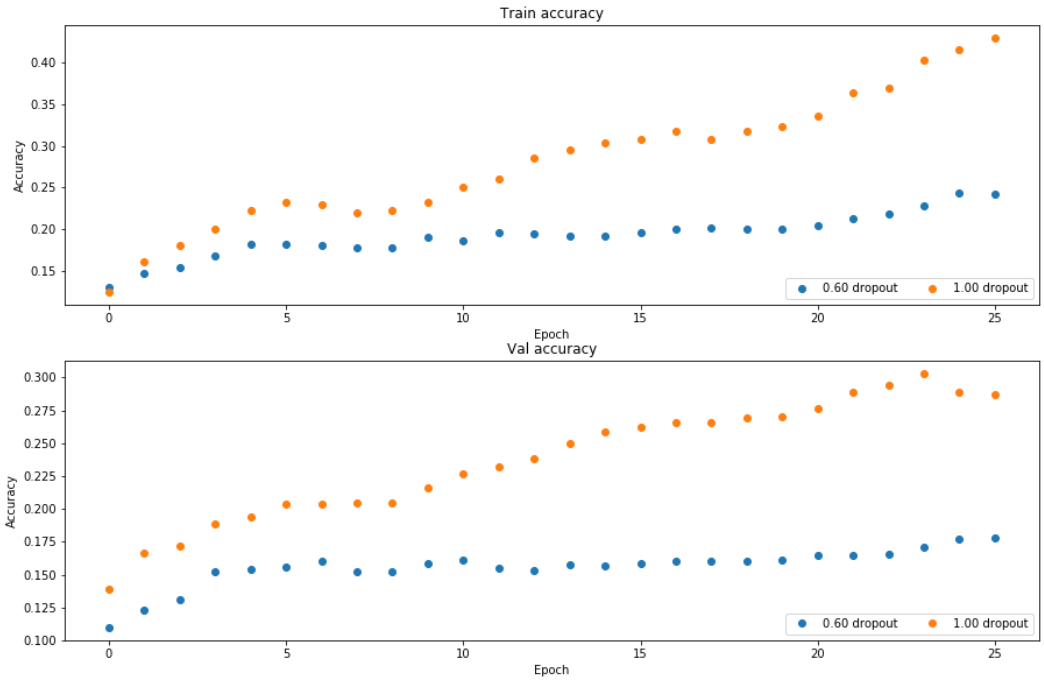
Dropout as a regularizer

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 = [0.6, 1.0]
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: 2.303100
(Epoch 0 / 25) train acc: 0.130000; val_acc: 0.110000
(Epoch 1 / 25) train acc: 0.146000; val_acc: 0.123000
(Epoch 2 / 25) train acc: 0.154000; val_acc: 0.131000
(Epoch 3 / 25) train acc: 0.168000; val_acc: 0.152000
(Epoch 4 / 25) train acc: 0.182000; val_acc: 0.154000
(Epoch 5 / 25) train acc: 0.182000; val_acc: 0.156000
(Epoch 6 / 25) train acc: 0.180000; val_acc: 0.160000
(Epoch 7 / 25) train acc: 0.178000; val_acc: 0.152000
(Epoch 8 / 25) train acc: 0.178000; val_acc: 0.152000
(Epoch 9 / 25) train acc: 0.190000; val_acc: 0.159000
(Epoch 10 / 25) train acc: 0.186000; val_acc: 0.161000
(Epoch 11 / 25) train acc: 0.196000; val_acc: 0.155000
(Epoch 12 / 25) train acc: 0.194000; val_acc: 0.153000
(Epoch 13 / 25) train acc: 0.192000; val_acc: 0.158000
(Epoch 14 / 25) train acc: 0.192000; val_acc: 0.157000
(Epoch 15 / 25) train acc: 0.196000; val_acc: 0.159000
(Epoch 16 / 25) train acc: 0.200000; val_acc: 0.160000
(Epoch 17 / 25) train acc: 0.202000; val_acc: 0.160000
(Epoch 18 / 25) train acc: 0.200000; val_acc: 0.160000
(Epoch 19 / 25) train acc: 0.200000; val_acc: 0.161000
(Epoch 20 / 25) train acc: 0.204000; val_acc: 0.165000
(Iteration 101 / 125) loss: 2.251923
(Epoch 21 / 25) train acc: 0.212000; val_acc: 0.165000
(Epoch 22 / 25) train acc: 0.218000; val_acc: 0.166000
(Epoch 23 / 25) train acc: 0.228000; val_acc: 0.171000
(Epoch 24 / 25) train acc: 0.244000; val_acc: 0.177000
(Epoch 25 / 25) train acc: 0.242000; val_acc: 0.178000
(Iteration 1 / 125) loss: 2.300607
(Epoch 0 / 25) train acc: 0.124000; val_acc: 0.139000
(Epoch 1 / 25) train acc: 0.160000; val_acc: 0.167000
(Epoch 2 / 25) train acc: 0.180000; val_acc: 0.172000
(Epoch 3 / 25) train acc: 0.200000; val_acc: 0.189000
(Epoch 4 / 25) train acc: 0.222000; val_acc: 0.194000
(Epoch 5 / 25) train acc: 0.232000; val_acc: 0.204000
(Epoch 6 / 25) train acc: 0.230000; val_acc: 0.204000
(Epoch 7 / 25) train acc: 0.220000; val_acc: 0.205000
(Epoch 8 / 25) train acc: 0.222000; val_acc: 0.205000
(Epoch 9 / 25) train acc: 0.232000; val_acc: 0.216000
(Epoch 10 / 25) train acc: 0.250000; val_acc: 0.227000
(Epoch 11 / 25) train acc: 0.260000; val_acc: 0.232000
(Epoch 12 / 25) train acc: 0.286000; val_acc: 0.238000
(Epoch 13 / 25) train acc: 0.296000; val_acc: 0.250000
(Epoch 14 / 25) train acc: 0.304000; val_acc: 0.259000
(Epoch 15 / 25) train acc: 0.308000; val_acc: 0.262000
(Epoch 16 / 25) train acc: 0.318000; val_acc: 0.266000
(Epoch 17 / 25) train acc: 0.308000; val_acc: 0.266000
(Epoch 18 / 25) train acc: 0.318000; val_acc: 0.269000
(Epoch 19 / 25) train acc: 0.324000; val_acc: 0.270000
(Epoch 20 / 25) train acc: 0.336000; val_acc: 0.276000
(Iteration 101 / 125) loss: 1.753858
(Epoch 21 / 25) train acc: 0.364000; val_acc: 0.289000
(Epoch 22 / 25) train acc: 0.370000; val_acc: 0.294000
(Epoch 23 / 25) train acc: 0.404000; val_acc: 0.303000
(Epoch 24 / 25) train acc: 0.416000; val_acc: 0.289000
(Epoch 25 / 25) train acc: 0.430000; val_acc: 0.287000
```

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 dropou
t' % 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:

Yes, it is. As we can see from our experiment result, training with more dropout rate achieves a better result than traning with less dropout rate. That is exactly what regularization is doing.

Final part of the assignment

Get over 55% 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%)) / 23%, 1) where if you get 55% or higher validation accuracy, you get full points.

In [8]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #

hidden_dims = [512, 256, 128, 64]
lr_decay = 0.9
learning_rate = 5e-3
weight_scale = 0.01
dropout = 0.2
reg = 0.0
update_rule = 'adam'

model = FullyConnectedNet(hidden_dims = hidden_dims, weight_scale = weig
ht_scale,
                          dropout=dropout, use_batchnorm=True, reg=reg)

solver = Solver(model, data,
                num_epochs=25, batch_size=1024,
                update_rule=update_rule,
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=lr_decay,
                verbose=True, print_every=500)
solver.train()

#print out the validation accuracy
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
'])))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
(Iteration 1 / 1175) loss: 2.305102
(Epoch 0 / 25) train acc: 0.221000; val_acc: 0.208000
(Epoch 1 / 25) train acc: 0.429000; val_acc: 0.408000
(Epoch 2 / 25) train acc: 0.445000; val_acc: 0.455000
(Epoch 3 / 25) train acc: 0.476000; val_acc: 0.505000
(Epoch 4 / 25) train acc: 0.519000; val_acc: 0.508000
(Epoch 5 / 25) train acc: 0.545000; val_acc: 0.519000
(Epoch 6 / 25) train acc: 0.584000; val_acc: 0.527000
(Epoch 7 / 25) train acc: 0.589000; val_acc: 0.535000
(Epoch 8 / 25) train acc: 0.594000; val_acc: 0.547000
(Epoch 9 / 25) train acc: 0.623000; val_acc: 0.553000
(Epoch 10 / 25) train acc: 0.630000; val_acc: 0.549000
(Iteration 501 / 1175) loss: 1.158378
(Epoch 11 / 25) train acc: 0.647000; val_acc: 0.551000
(Epoch 12 / 25) train acc: 0.644000; val_acc: 0.545000
(Epoch 13 / 25) train acc: 0.653000; val_acc: 0.556000
(Epoch 14 / 25) train acc: 0.668000; val_acc: 0.558000
(Epoch 15 / 25) train acc: 0.664000; val_acc: 0.564000
(Epoch 16 / 25) train acc: 0.678000; val_acc: 0.563000
(Epoch 17 / 25) train acc: 0.708000; val_acc: 0.558000
(Epoch 18 / 25) train acc: 0.721000; val_acc: 0.573000
(Epoch 19 / 25) train acc: 0.677000; val_acc: 0.573000
(Epoch 20 / 25) train acc: 0.716000; val_acc: 0.571000
(Epoch 21 / 25) train acc: 0.704000; val_acc: 0.570000
(Iteration 1001 / 1175) loss: 0.964139
(Epoch 22 / 25) train acc: 0.721000; val_acc: 0.574000
(Epoch 23 / 25) train acc: 0.736000; val_acc: 0.584000
(Epoch 24 / 25) train acc: 0.713000; val_acc: 0.575000
(Epoch 25 / 25) train acc: 0.751000; val_acc: 0.566000
Validation set accuracy: 0.584
Test set accuracy: 0.589
```

```
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 TwoLayerNet(object):
    """
    A two-layer fully-connected neural network with ReLU nonlinearity and
    softmax loss that uses a modular layer design. We assume an input dimension
    of D, a hidden dimension of H, and perform classification over C classes.

    The architecure should be affine - relu - affine - softmax.

    Note that this class does not implement gradient descent; instead, it
    will interact with a separate Solver object that is responsible for running
    optimization.

    The learnable parameters of the model are stored in the dictionary
    self.params that maps parameter names to numpy arrays.
    """

    def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
                 dropout=0, weight_scale=1e-3, reg=0.0):
        """
        Initialize a new network.

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

        # ============================================================ #
        # YOUR CODE HERE:
        #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
        #   self.params['W2'], self.params['b1'] and self.params['b2']. The
        #   biases are initialized to zero and the weights are initialized
        #   so that each parameter has mean 0 and standard deviation weight_scale.
        #   The dimensions of W1 should be (input_dim, hidden_dim) and the
        #   dimensions of W2 should be (hidden_dims, num_classes)
        # ============================================================ #
        self.params['W1'] = np.random.randn(input_dim, hidden_dims) * weight_scale
        self.params['W2'] = np.random.randn(hidden_dims, num_classes) *
weight_scale
        self.params['b1'] = np.zeros(hidden_dims)
        self.params['b2'] = np.zeros(num_classes)


        # ============================================================ #
```

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

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

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

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

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

        # ============================================================ #
        # YOUR CODE HERE:
        #   Implement the forward pass of the two-layer neural network. Store
        #   the class scores as the variable 'scores'.  Be sure to use the layers
        #   you prior implemented.
        # ============================================================ #
        out1, cache1 = affine_relu_forward(X, self.params['W1'], self.params['b1'])
        scores, cache2 = affine_forward(out1, self.params['W2'], self.params['b2'])
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #

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

        loss, grads = 0, {}
        # ============================================================ #
        # YOUR CODE HERE:
        #   Implement the backward pass of the two-layer neural net.  Store
        #   the loss as the variable 'loss' and store the gradients in the
        #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
        #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
        #   i.e., grads[k] holds the gradient for self.params[k].
        #
        #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
        #   for each W.  Be sure to include the 0.5 multiplying factor to
        #   match our implementation.
        #
        #   And be sure to use the layers you prior implemented.
        # ============================================================ #
        loss,dscores = softmax_loss(scores, y)
        loss += 0.5 * self.reg * np.sum(self.params['W1'] ** 2) + 0.5 * self.reg *
np.sum(self.params['W2'] ** 2)
        dx2,grads['W2'],grads['b2'] = affine_backward(dscores,cache2)
        dx1,grads['W1'],grads['b1'] = affine_relu_backward(dx2,cache1)
        grads['W2'] += self.reg * self.params['W2']
        grads['W1'] += self.reg * self.params['W1']
        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
```

```
        return loss, grads


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=1
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 < 1
        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 do batch normalize the output scores.
        # ============================================================ #
        arbi_input_dim = input_dim
```

```
        for i, hd in enumerate(hidden_dims):
            arbiW = 'W%d' % (i + 1)
            arbib = 'b%d' % (i + 1)

            self.params[arbiW] = np.random.randn(arbi_input_dim,  hd) *
weight_scale
            self.params[arbib] = np.zeros(hd)
            # Initialize
            if self.use_batchnorm:
                arbiGamma = 'gamma%d' % (i + 1)
                arbiBeta = 'beta%d' % (i + 1)
                self.params[arbiGamma] = np.ones(hd)
                self.params[arbiBeta] = np.zeros(hd)

            arbi_input_dim = hd

        self.params['W%d' % self.num_layers] = np.random.randn(arbi_input_dim,
num_classes) * weight_scale
        self.params['b%d' % self.num_layers] = np.zeros(num_classes)

        # ============================================================ #
        # 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.
        # ============================================================ #
        fc_cache = {}
        relu_cache = {}
        bn_cache = {}
        dropout_cache = {}
        batch_size = X.shape[0]

        X = np.reshape(X, [batch_size, -1])

        for i in range(self.num_layers-1):
            fc_act, fc_cache['%d' % (i+1)] = affine_forward(X, self.params['W%d' %
(i+1)], self.params['b%d' % (i+1)])
            if self.use_batchnorm:
                bn_act, bn_cache['%d' % (i+1)] = batchnorm_forward(fc_act,
self.params['gamma%d' % (i+1)], self.params['beta%d' % (i+1)], self.bn_params[i])
                relu_act, relu_cache['%d' % (i+1)] = relu_forward(bn_act)
            else:
                relu_act, relu_cache['%d' % (i+1)] = relu_forward(fc_act)
            if self.use_dropout:
                relu_act, dropout_cache['%d' % (i+1)] = dropout_forward(relu_act,
self.dropout_param)

            X = relu_act.copy()  # Result of one pass through the affine-relu
block.

        # Output layer is FC layer without relu.
        scores, final_cache = affine_forward(X, self.params['W'+str
(self.num_layers)], self.params['b'+str(self.num_layers)])

        # ============================================================ #
        # 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.
        # ============================================================ #
        loss, dsm = softmax_loss(scores, y)
        loss += 0.5 * self.reg * (np.sum(np.square(self.params['W%d' %
(self.num_layers)])))
```

```
        dx_last, dw_last, db_last = affine_backward(dsm, final_cache)

        grads['W%d' % (self.num_layers)] = dw_last + self.reg*self.params['W%d' %
(self.num_layers)]
        grads['b%d' % (self.num_layers)] = db_last
        for i in range(self.num_layers - 1, 0, -1):
            if self.use_dropout:
                dx_last = dropout_backward(dx_last, dropout_cache['%d' % (i)])

            drelu = relu_backward(dx_last, relu_cache['%d' % (i)])
            if self.use_batchnorm:
                dbatchnorm, dgamma, dbeta = batchnorm_backward(drelu, bn_cache['%
d' % (i)])
                dx_last, dw_last, db_last = affine_backward(dbatchnorm, fc_cache['%
d' % (i)])
                grads['beta%d' % (i)] = dbeta
                grads['gamma%d' % (i)] = dgamma
            else:
                dx_last, dw_last, db_last = affine_backward(drelu, fc_cache['%d' %
(i)])

            grads['W%d' % (i)] = dw_last + self.reg * self.params['W%d' % (i)]
            grads['b%d' % (i)] = db_last

            loss += 0.5 * self.reg * (np.sum(np.square(self.params['W%d' % (i)])))


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

        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)
    """
    out = None
    # ============================================================ #
    # 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.
    # ============================================================ #
    N = x.shape[0]
    D = w.shape[0]
    x_inp = x.reshape(N,D)
    out = np.dot(x_inp,w) + b


    # ============================================================ #
    # 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: 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:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
```

```
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # Notice:
    #   dout is N x M
    #   dx should be N x d1 x ... x dk; it relates to dout through multiplication
with w, which is D x M
    #   dw should be D x M; it relates to dout through multiplication with x,
which is N x D after reshaping
    #   db should be M; it is just the sum over dout examples
    # ============================================================= #
    N = x.shape[0]
    D = w.shape[0]
    x_inp = x.reshape(N,D)

    dx_inp = np.dot(dout,w.T)
    dx = dx_inp.reshape(x.shape)
    dw = np.dot(x_inp.T,dout)
    db = np.dot(dout.T,np.ones(N))


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

    return dx, dw, db

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

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ============================================================= #

    out = np.maximum(0,x)
    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #

    cache = x
    return out, cache


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

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
```

```
    - dx: Gradient with respect to x
    """
    x = cache

    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the ReLU backward pass
    # ============================================================= #
    dx = np.asarray(dout)
    dx[x<=0] = 0

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

    return dx

def batchnorm_forward(x, gamma, beta, bn_param):
    """
    Forward pass for batch normalization.

    During training the sample mean and (uncorrected) sample variance are
    computed from minibatch statistics and used to normalize the incoming data.
    During training we also keep an exponentially decaying running mean of the mean
    and variance of each feature, and these averages are used to normalize data
    at test-time.

    At each timestep we update the running averages for mean and variance using
    an exponential decay based on the momentum parameter:

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    Note that the batch normalization paper suggests a different test-time
    behavior: they compute sample mean and variance for each feature using a
    large number of training images rather than using a running average. For
    this implementation we have chosen to use running averages instead since
    they do not require an additional estimation step; the torch7 implementation
    of batch normalization also uses running averages.

    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == 'train':
```

```
        # ============================================================ #
        # YOUR CODE HERE:
        #   A few steps here:
        #     (1) Calculate the running mean and variance of the minibatch.
        #     (2) Normalize the activations with the running mean and variance.
        #     (3) Scale and shift the normalized activations.  Store this
        #         as the variable 'out'
        #     (4) Store any variables you may need for the backward pass in
        #         the 'cache' variable.
        # ============================================================ #

        arbiBatch_mean = np.mean(x, axis = 0)
        arbiBatch_var = np.var(x, axis = 0)

        running_mean = momentum * running_mean + (1 - momentum) * arbiBatch_mean
        running_var = momentum * running_var + (1 - momentum) * arbiBatch_var
        bn_param['running_mean'] = running_mean
        bn_param['running_var'] = running_var

        arbiBatch_norm = (x - arbiBatch_mean) / np.sqrt(arbiBatch_var + eps)
        out = gamma * arbiBatch_norm + beta

        cache = {
            'x_minus_mean': (x - arbiBatch_mean),
            'arbiBatch_norm': arbiBatch_norm,
            'gamma': gamma,
            'inv_var': 1.0 / np.sqrt(arbiBatch_var + eps),
            'sqrt_var': np.sqrt(arbiBatch_var + eps)
        }

        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
    elif mode == 'test':
        # ============================================================ #
        # YOUR CODE HERE:
        #   Calculate the testing time normalized activation.  Normalize using
        #   the running mean and variance, and then scale and shift appropriately.
        #   Store the output as 'out'.
        # ============================================================ #

        out = (gamma * x / (np.sqrt(running_var + eps))) + (beta - (gamma *
running_mean) / np.sqrt(running_var + eps))

        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    return out, cache

def batchnorm_backward(dout, cache):
    """
    Backward pass for batch normalization.

    For this implementation, you should write out a computation graph for
    batch normalization on paper and propagate gradients backward through
    intermediate nodes.

    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
```

```
    - cache: Variable of intermediates from batchnorm_forward.

    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None

    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
    # ============================================================= #
    N,D = dout.shape
    x_minus_mean = cache.get('x_minus_mean')
    arbiBatch_norm = cache.get('arbiBatch_norm')
    gamma = cache.get('gamma')
    inv_var = cache.get('inv_var')
    sqrt_var = cache.get('sqrt_var')

    dxhat = dout * gamma
    dxmu1 = dxhat * inv_var
    dinv_var = np.sum(dxhat * x_minus_mean, axis = 0)
    dsqrt_var = dinv_var * (-1.0 / sqrt_var**2)
    dvar = dsqrt_var * 0.5 * (1 / sqrt_var)
    dsq = (1.0 / N) * dvar * np.ones_like(dout)
    dxmu2 = dsq * 2 * x_minus_mean
    dx1 = dxmu1 + dxmu2
    dmu = -1 * np.sum(dxmu1 + dxmu2, axis = 0)
    dx2 = (1 / N) * dmu * np.ones_like(dout)
    dx = dx1 + dx2

    dbeta = np.sum(dout, axis = 0)
    dgamma = np.sum(dout * arbiBatch_norm, axis = 0)

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

    return dx, dgamma, dbeta

def dropout_forward(x, dropout_param):
    """
    Performs the forward pass for (inverted) dropout.

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes this
        function deterministic, which is needed for gradient checking but not in
        real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
      mask that was used to multiply the input; in test mode, mask is None.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])

    mask = None
    out = None
```

```
    if mode == 'train':
        # ============================================================= #
        # YOUR CODE HERE:
        #   Implement the inverted dropout forward pass during training time.
        #   Store the masked and scaled activations in out, and store the
        #   dropout mask as the variable mask.
        # ============================================================= #


        mask = (np.random.random_sample(x.shape) >= p)
        dropout_factor = 1 / (1 - p)
        mask = mask * dropout_factor

        out = x * mask
        # ============================================================= #
        # END YOUR CODE HERE
        # ============================================================= #

    elif mode == 'test':

        # ============================================================= #
        # YOUR CODE HERE:
        #   Implement the inverted dropout forward pass during test time.
        # ============================================================= #


        out = x
        # ============================================================= #
        # END YOUR CODE HERE
        # ============================================================= #

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache

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

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

    dx = None
    if mode == 'train':
        # ============================================================= #
        # YOUR CODE HERE:
        #   Implement the inverted dropout backward pass during training time.
        # ============================================================= #


        dx = dout * mask
        # ============================================================= #
        # END YOUR CODE HERE
        # ============================================================= #
    elif mode == 'test':
        # ============================================================= #
        # YOUR CODE HERE:
        #   Implement the inverted dropout backward pass during test time.
        # ============================================================= #
```

```
        dx = dout
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
    return dx

def svm_loss(x, y):
    """
    Computes the loss and gradient using for multiclass SVM classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    N = x.shape[0]
    correct_class_scores = x[np.arange(N), y]
    margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
    margins[np.arange(N), y] = 0
    loss = np.sum(margins) / N
    num_pos = np.sum(margins > 0, axis=1)
    dx = np.zeros_like(x)
    dx[margins > 0] = 1
    dx[np.arange(N), y] -= num_pos
    dx /= N
    return loss, dx


def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
      for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```