ECE C147, Winter 2020

Homework 4

Submitted by Guy Ohayon

All of the relevant code is included at the end of this document.

1 Implementing different optimizers for a fully connected network

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 [20]:
```
## 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_numeri
cal_gradient_array
from cs231n.solver import Solver

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

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

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

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

In [21]:
```
# 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)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
y_test: (1000,)
X_test: (1000, 3, 32, 32)
y_train: (49000,)
```

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 [22]:
```
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: 9.769848888397517e-10


If affine_backward is working, error should be less than 1e-9::
dx error: 9.825668876022365e-09
dw error: 9.109886583897834e-10
db error: 1.0324487413514516e-10


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: 3.2756391206508e-12


If affine_relu_forward and affine_relu_backward are working, error sho
uld be less than 1e-9::
dx error: 2.8283154522945933e-10
dw error: 1.4917340422768562e-09
db error: 2.5479972508511626e-11


Running check with reg = 0
Initial loss: 2.305586415971783
W1 relative error: 3.1672462718241966e-07
W2 relative error: 2.264803018868459e-07
W3 relative error: 4.4554812249210325e-07
b1 relative error: 7.771674906015045e-09
b2 relative error: 5.7628378726414716e-09
b3 relative error: 1.226876752734328e-10
Running check with reg = 3.14
Initial loss: 7.079562020283974
W1 relative error: 1.0130826921822634e-08
W2 relative error: 3.772738942736491e-08
W3 relative error: 3.2000059239676934e-09
b1 relative error: 7.770018658295104e-09
b2 relative error: 1.5649207430998137e-08
b3 relative error: 1.2174012715113925e-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 [23]:
```
from nndl.optim import sgd_momentum

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

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

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

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

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

SGD + Nesterov momentum

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

In [24]:
```
from nndl.optim import sgd_nesterov_momentum

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

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

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

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

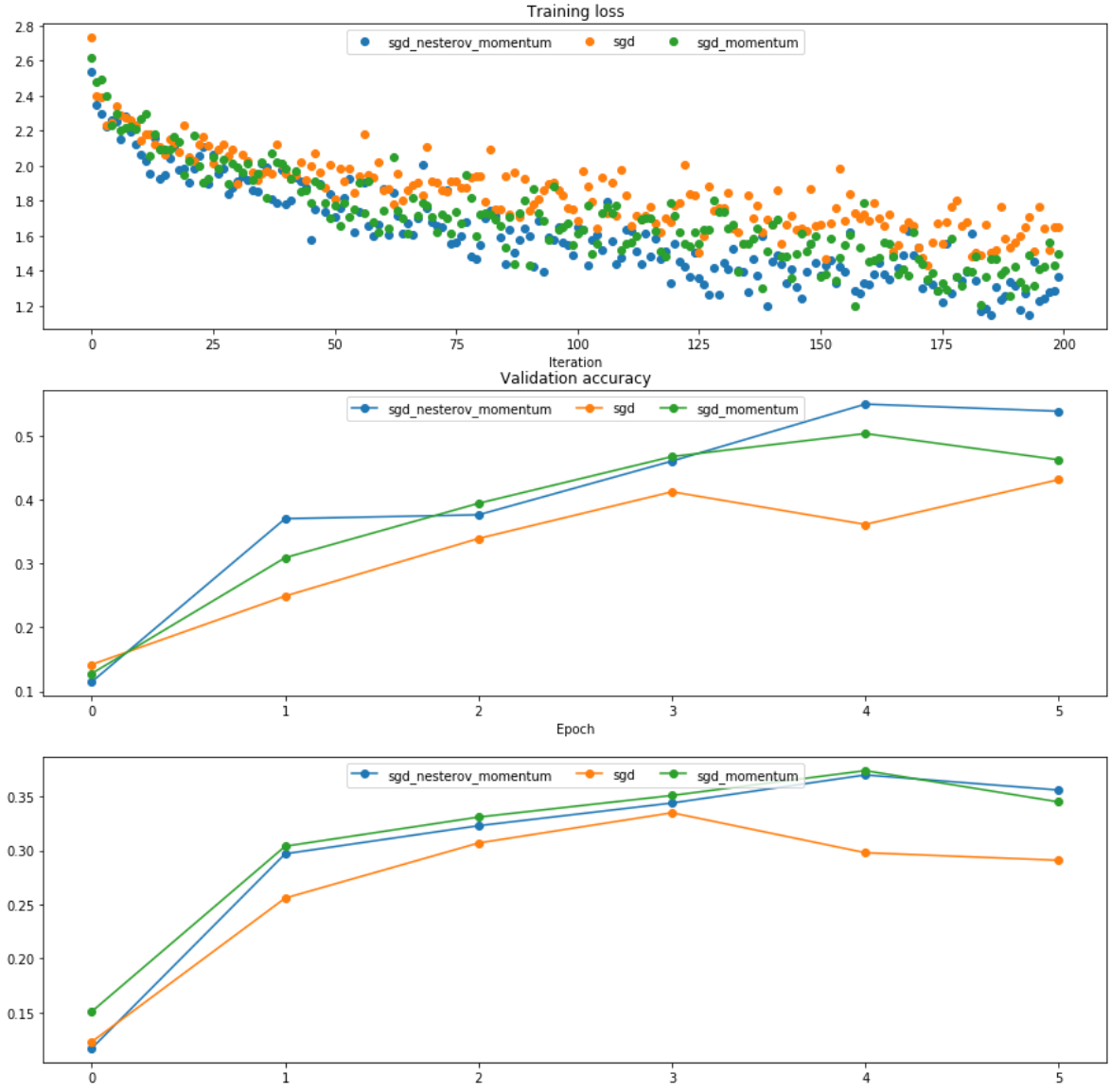
```
next_w error: 1.0875186845081027e-08
velocity error: 4.269287743278663e-09
```

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 [25]:
```
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 [27]:
```
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 [28]:
```
# 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'])))
```

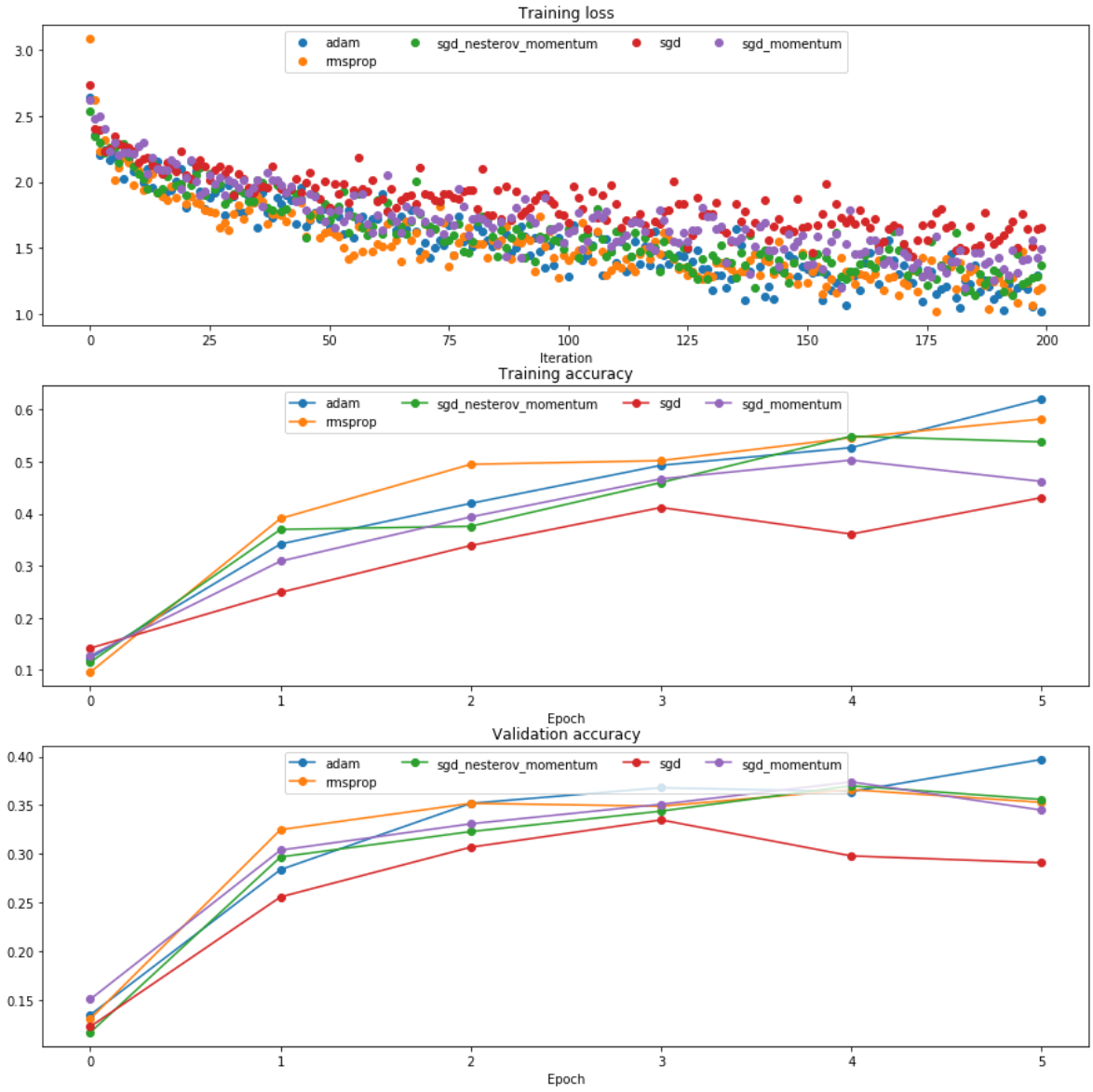
```
next_w error: 1.1395691798535431e-07
a error: 4.208314038113071e-09
v error: 4.214963193114416e-09
```

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 [29]:
```
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 [31]:
```
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=False)

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: 2.325880
(Epoch 0 / 10) train acc: 0.137000; val_acc: 0.129000
(Iteration 51 / 4900) loss: 1.814975
(Iteration 101 / 4900) loss: 1.683150
(Iteration 151 / 4900) loss: 1.718433
(Iteration 201 / 4900) loss: 1.719244
(Iteration 251 / 4900) loss: 1.632982
(Iteration 301 / 4900) loss: 1.571701
(Iteration 351 / 4900) loss: 1.759656
(Iteration 401 / 4900) loss: 1.675472
(Iteration 451 / 4900) loss: 1.320238
(Epoch 1 / 10) train acc: 0.420000; val_acc: 0.439000
(Iteration 501 / 4900) loss: 1.646635
(Iteration 551 / 4900) loss: 1.465340
(Iteration 601 / 4900) loss: 1.322049
(Iteration 651 / 4900) loss: 1.552716
(Iteration 701 / 4900) loss: 1.549162
(Iteration 751 / 4900) loss: 1.459924
(Iteration 801 / 4900) loss: 1.530061
(Iteration 851 / 4900) loss: 1.344491
(Iteration 901 / 4900) loss: 1.674584
(Iteration 951 / 4900) loss: 1.441113
(Epoch 2 / 10) train acc: 0.483000; val_acc: 0.469000
(Iteration 1001 / 4900) loss: 1.200922
(Iteration 1051 / 4900) loss: 1.503927
(Iteration 1101 / 4900) loss: 1.292281
(Iteration 1151 / 4900) loss: 1.560023
(Iteration 1201 / 4900) loss: 1.383492
(Iteration 1251 / 4900) loss: 1.470189
(Iteration 1301 / 4900) loss: 1.423932
(Iteration 1351 / 4900) loss: 1.314237
(Iteration 1401 / 4900) loss: 1.378785
(Iteration 1451 / 4900) loss: 1.379929
(Epoch 3 / 10) train acc: 0.531000; val_acc: 0.460000
(Iteration 1501 / 4900) loss: 1.366102
(Iteration 1551 / 4900) loss: 1.307615
(Iteration 1601 / 4900) loss: 1.354900
(Iteration 1651 / 4900) loss: 1.275502
(Iteration 1701 / 4900) loss: 1.477173
(Iteration 1751 / 4900) loss: 1.500983
(Iteration 1801 / 4900) loss: 1.242940
(Iteration 1851 / 4900) loss: 1.531579
(Iteration 1901 / 4900) loss: 1.435962
(Iteration 1951 / 4900) loss: 1.209449
(Epoch 4 / 10) train acc: 0.558000; val_acc: 0.502000
(Iteration 2001 / 4900) loss: 1.354923
(Iteration 2051 / 4900) loss: 1.247311
(Iteration 2101 / 4900) loss: 1.226677
(Iteration 2151 / 4900) loss: 1.445371
(Iteration 2201 / 4900) loss: 1.456219
(Iteration 2251 / 4900) loss: 1.456376
(Iteration 2301 / 4900) loss: 1.117906
(Iteration 2351 / 4900) loss: 1.175267
(Iteration 2401 / 4900) loss: 1.138969
(Epoch 5 / 10) train acc: 0.553000; val_acc: 0.510000
(Iteration 2451 / 4900) loss: 1.209894
(Iteration 2501 / 4900) loss: 1.215077
```

```
(Iteration 2551 / 4900) loss: 1.154377
(Iteration 2601 / 4900) loss: 1.052938
(Iteration 2651 / 4900) loss: 1.211362
(Iteration 2701 / 4900) loss: 1.260336
(Iteration 2751 / 4900) loss: 1.199746
(Iteration 2801 / 4900) loss: 1.091490
(Iteration 2851 / 4900) loss: 0.893251
(Iteration 2901 / 4900) loss: 0.909857
(Epoch 6 / 10) train acc: 0.605000; val_acc: 0.509000
(Iteration 2951 / 4900) loss: 1.112212
(Iteration 3001 / 4900) loss: 0.923820
(Iteration 3051 / 4900) loss: 1.087975
(Iteration 3101 / 4900) loss: 1.231528
(Iteration 3151 / 4900) loss: 1.150509
(Iteration 3201 / 4900) loss: 1.111537
(Iteration 3251 / 4900) loss: 1.099435
(Iteration 3301 / 4900) loss: 1.151229
(Iteration 3351 / 4900) loss: 1.096382
(Iteration 3401 / 4900) loss: 1.205294
(Epoch 7 / 10) train acc: 0.589000; val_acc: 0.510000
(Iteration 3451 / 4900) loss: 1.165627
(Iteration 3501 / 4900) loss: 1.114332
(Iteration 3551 / 4900) loss: 1.094929
(Iteration 3601 / 4900) loss: 0.990226
(Iteration 3651 / 4900) loss: 0.930244
(Iteration 3701 / 4900) loss: 1.172076
(Iteration 3751 / 4900) loss: 1.026694
(Iteration 3801 / 4900) loss: 1.144197
(Iteration 3851 / 4900) loss: 1.006173
(Iteration 3901 / 4900) loss: 0.912555
(Epoch 8 / 10) train acc: 0.634000; val_acc: 0.529000
(Iteration 3951 / 4900) loss: 0.999805
(Iteration 4001 / 4900) loss: 1.026308
(Iteration 4051 / 4900) loss: 0.983128
(Iteration 4101 / 4900) loss: 0.928602
(Iteration 4151 / 4900) loss: 1.159897
(Iteration 4201 / 4900) loss: 0.984422
(Iteration 4251 / 4900) loss: 0.981149
(Iteration 4301 / 4900) loss: 0.913680
(Iteration 4351 / 4900) loss: 0.874465
(Iteration 4401 / 4900) loss: 0.871531
(Epoch 9 / 10) train acc: 0.631000; val_acc: 0.529000
(Iteration 4451 / 4900) loss: 0.834450
(Iteration 4501 / 4900) loss: 0.965575
(Iteration 4551 / 4900) loss: 0.729105
(Iteration 4601 / 4900) loss: 0.928025
(Iteration 4651 / 4900) loss: 0.898482
(Iteration 4701 / 4900) loss: 1.073172
(Iteration 4751 / 4900) loss: 0.916686
(Iteration 4801 / 4900) loss: 0.735491
(Iteration 4851 / 4900) loss: 0.967441
(Epoch 10 / 10) train acc: 0.659000; val_acc: 0.532000
```

In [32]:
```
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_te
st'])))
```

```
Validation set accuracy: 0.532
Test set accuracy: 0.531
```

2 Implementing batch normalization for a fully connected network

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_numeri
cal_gradient_array
from cs231n.solver import Solver

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

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

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

In [2]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

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 [11]:
```
# 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':
'train'})
print('  mean: ', a_norm.mean(axis=0))
print('  std: ', a_norm.std(axis=0))

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

```
Before batch normalization:
  means:  [ 36.62133438 -30.57944044 -14.3934147 ]
  stds:  [ 29.30312306  29.89680859  35.60057534]
After batch normalization (gamma=1, beta=0)
  mean:  [ -5.89528426e-16   1.18689780e-16   1.29896094e-16]
  std:  [ 0.99999999  0.99999999  1.        ]
After batch normalization (nontrivial gamma, beta)
  means:  [ 11.  12.  13.]
  stds:  [ 0.99999999  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 [12]:
```
# 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:  [-0.00628248 -0.03568937 -0.16954105]
  stds:  [ 1.12922252  0.95989921  1.12391224]
```

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 [15]:
```
# 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:  9.06199618046e-10
dgamma error:  8.78981054069e-11
dbeta error:  6.64112110292e-11
```

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 [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 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.flo
at64,
                              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], verb
ose=False, 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:  2.41908700045
W1 relative error: 7.594388353742188e-06
W2 relative error: 2.641774473466689e-06
W3 relative error: 4.918391341136713e-10
b1 relative error: 2.1316282072803006e-06
b2 relative error: 6.661338147750939e-07
b3 relative error: 8.210762647336403e-11
beta1 relative error: 5.2596656922790215e-09
beta2 relative error: 1.4590830145168932e-09
gamma1 relative error: 4.641588791433469e-09
gamma2 relative error: 1.7604919207261517e-09


Running check with reg =  3.14
Initial loss:  7.06586957069
W1 relative error: 3.684672277175489e-08
W2 relative error: 5.065906283068243e-06
W3 relative error: 7.468461679205003e-09
b1 relative error: 0.0044408587918098865
b2 relative error: 8.881784197001252e-08
b3 relative error: 1.925522417528727e-10
beta1 relative error: 1.769122223955703e-08
beta2 relative error: 5.190884342593366e-08
gamma1 relative error: 1.401437805098469e-08
gamma2 relative error: 6.420523210451984e-09
```

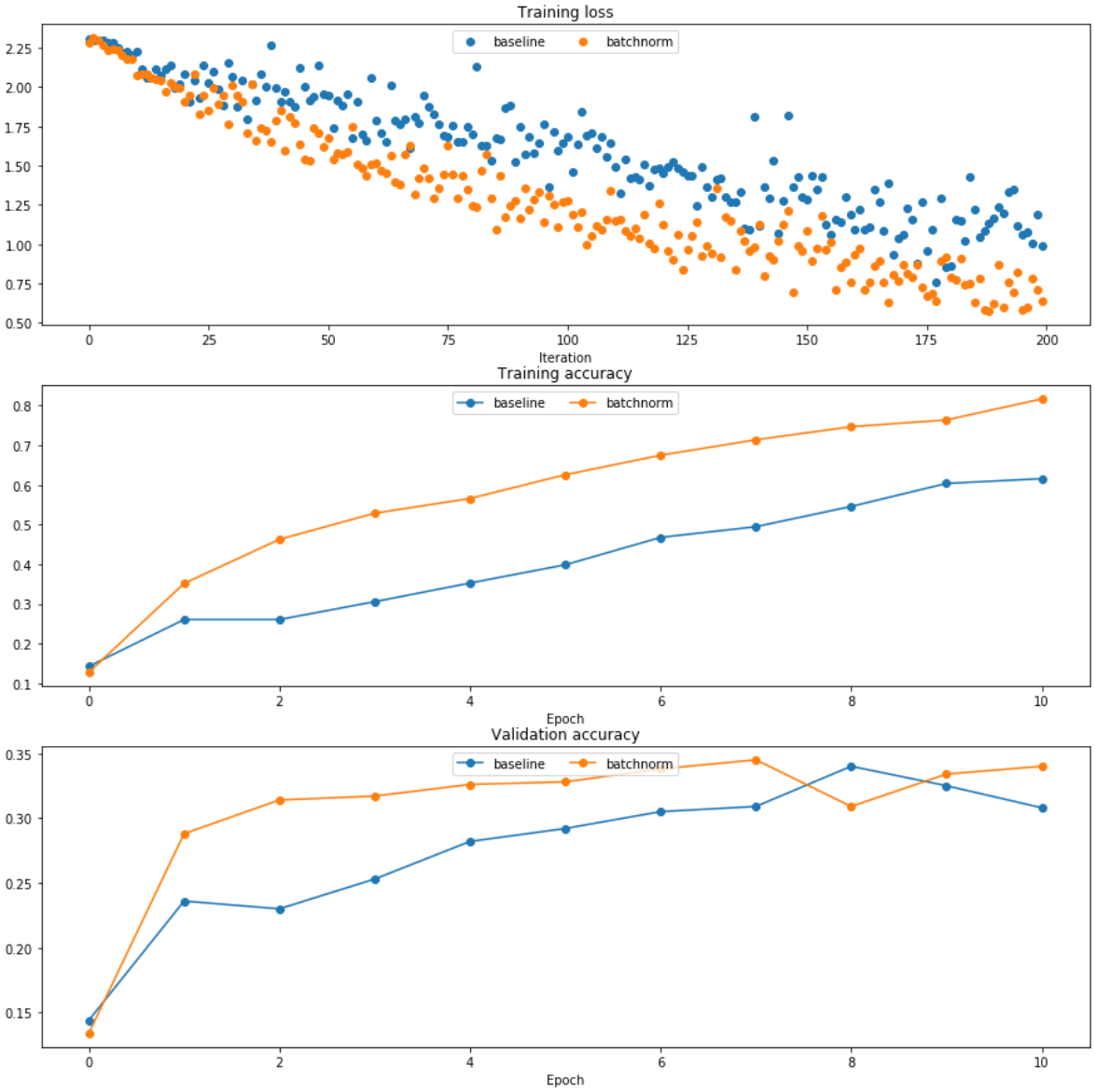
Training a deep fully connected network with batch normalization.

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

In [52]:
```
# 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, 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=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: 2.283746
(Epoch 0 / 10) train acc: 0.128000; val_acc: 0.134000
(Epoch 1 / 10) train acc: 0.352000; val_acc: 0.288000
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.314000
(Epoch 3 / 10) train acc: 0.529000; val_acc: 0.317000
(Epoch 4 / 10) train acc: 0.566000; val_acc: 0.326000
(Epoch 5 / 10) train acc: 0.626000; val_acc: 0.328000
(Epoch 6 / 10) train acc: 0.675000; val_acc: 0.338000
(Epoch 7 / 10) train acc: 0.714000; val_acc: 0.345000
(Epoch 8 / 10) train acc: 0.747000; val_acc: 0.309000
(Epoch 9 / 10) train acc: 0.764000; val_acc: 0.334000
(Epoch 10 / 10) train acc: 0.817000; val_acc: 0.340000
(Iteration 1 / 200) loss: 2.303005
(Epoch 0 / 10) train acc: 0.143000; val_acc: 0.144000
(Epoch 1 / 10) train acc: 0.261000; val_acc: 0.236000
(Epoch 2 / 10) train acc: 0.261000; val_acc: 0.230000
(Epoch 3 / 10) train acc: 0.306000; val_acc: 0.253000
(Epoch 4 / 10) train acc: 0.353000; val_acc: 0.282000
(Epoch 5 / 10) train acc: 0.399000; val_acc: 0.292000
(Epoch 6 / 10) train acc: 0.468000; val_acc: 0.305000
(Epoch 7 / 10) train acc: 0.495000; val_acc: 0.309000
(Epoch 8 / 10) train acc: 0.546000; val_acc: 0.340000
(Epoch 9 / 10) train acc: 0.604000; val_acc: 0.325000
(Epoch 10 / 10) train acc: 0.616000; val_acc: 0.308000
```

In [53]:
```
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.

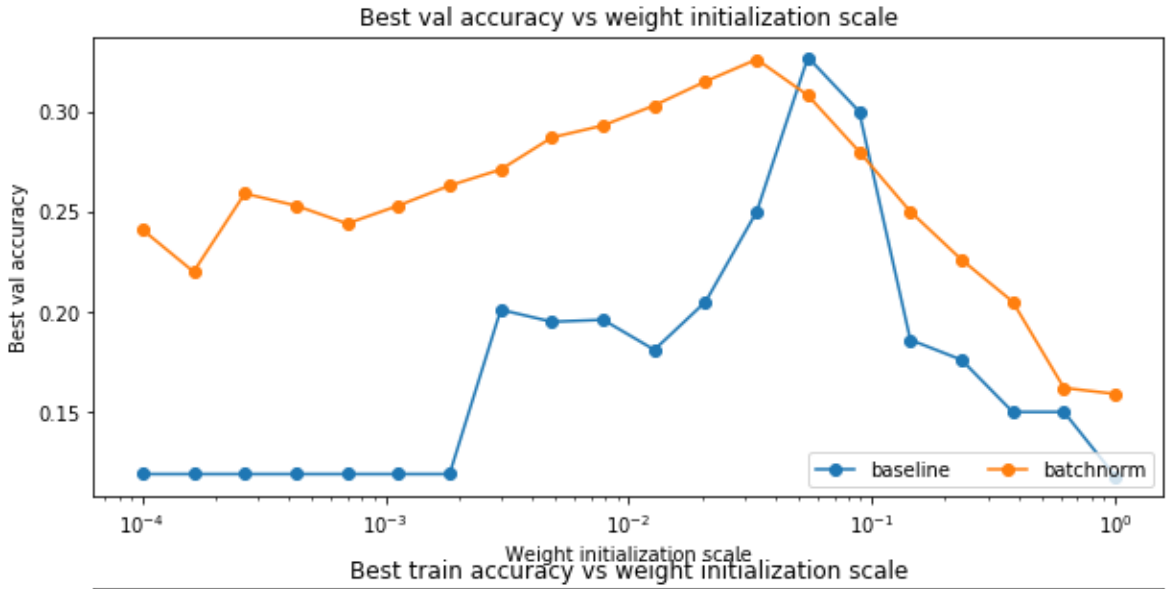
In [57]:
```
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]

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_scal
es)))
    bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scal
e, use_batchnorm=True)
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,
use_batchnorm=False)

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

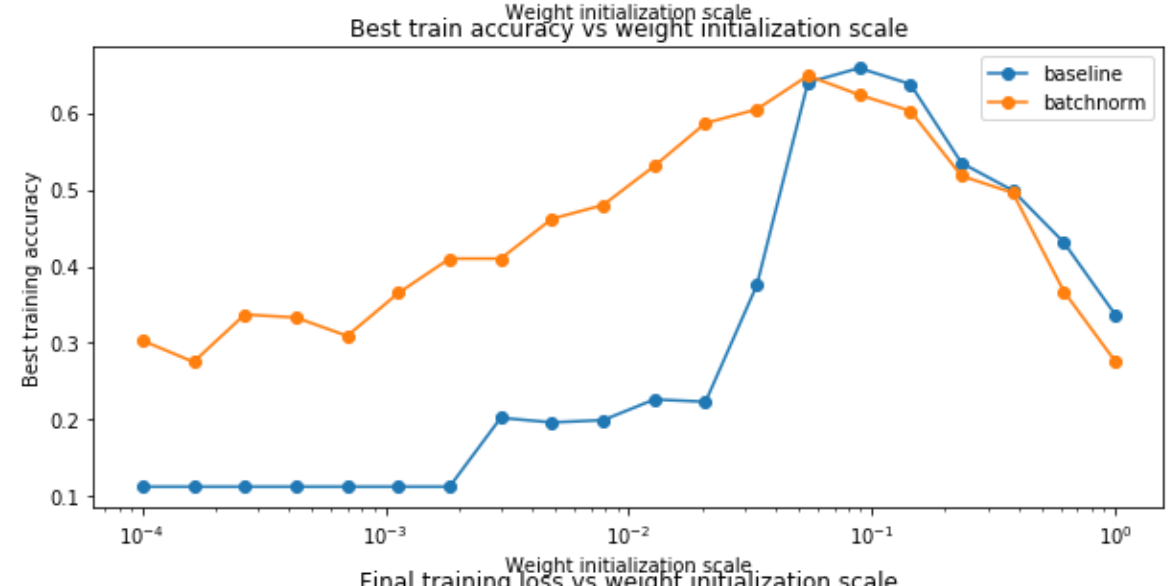
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale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

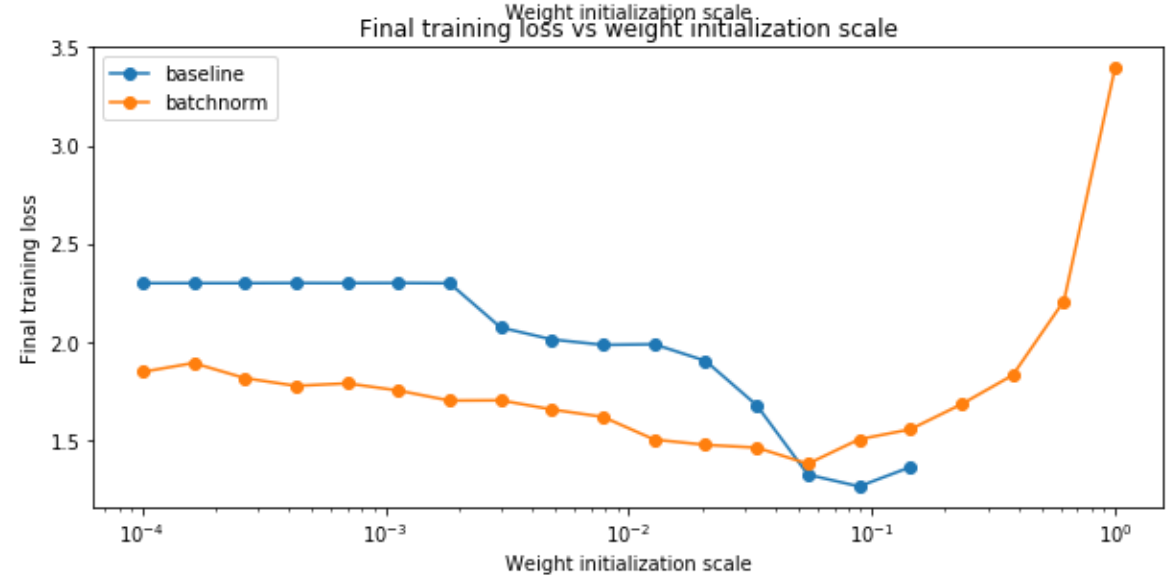
```
/home/ohayonguy/Courses/ECE_C147/HW4/nndl/layers.py:410: RuntimeWarnin
g: divide by zero encountered in log
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
```

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

In [55]:
```
# 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='batchnor
m')
plt.legend()

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

We can see that the weight initialization scale affects the performance of our model whether if we use batch normalization or not. In addition, batch normalization tends to improve training and validation accuracy for most weight initialization scales that were tested.

The statistics of the features that propagates forward (inputs for each layer) can vary a lot between iterations because when we forward propagate a minibatch, the statistics can vary from minibatch to minibatch. Thus, during learning, the weights of each layer would need to constantly adapt to different statistics of the data (minibatch), which would limit the training capacity of the model to just learn how to seperate the data, and thus would slow down learning and possibly limit the accuracy of the model. Rather, it's better that the gradients would update the weights in a way that's focused on seperating the data (which remains with the roughly the same statistics) and creating useful features for these statistics. This is exactly what batch normalization is doing, and so it makes sense that batch normalization helps the model arrive to a better accuracy, both for training and validation.

We can observe that batch normalization does the job it's supposed to do. When the weight initialization scale is big, batch norm prevents the gradients from being too large and thus prevents the model from divergence. Also, when the initialization scale is small, batch norm prevents the gradients from being too small and thus helps training to keep on going even though the weights are small initially.

3 Implementing dropout for a fully connected network, and optimizing it

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_numeri
cal_gradient_array
from cs231n.solver import Solver

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

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

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

In [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)
X_test: (1000, 3, 32, 32)
y_val: (1000,)
X_val: (1000, 3, 32, 32)
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 [4]:
```
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).me
an())
    print('Fraction of test-time output set to zero: ', (out_test == 0
).mean())
```

```
Running tests with p =  0.3
Mean of input:  9.99853206456
Mean of train-time output:  10.0156191475
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.699472
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.6
Mean of input:  9.99853206456
Mean of train-time output:  10.0040205996
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.399612
Fraction of test-time output set to zero:  0.0
Running tests with p =  0.75
Mean of input:  9.99853206456
Mean of train-time output:  10.00547198
Mean of test-time output:  9.99853206456
Fraction of train-time output set to zero:  0.249576
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 [6]:
```
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.8, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,
dropout_param)[0], x, dout)

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

```
dx relative error:  1.89289661024e-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 [10]:
```
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], verb
ose=False, h=1e-5)
        print('{} relative error: {}'.format(name, rel_error(grad_num,
grads[name])))
    print('\n')
```

```
Running check with dropout =  0.5
Initial loss:  2.30355723855
W1 relative error: 4.566234110910478e-08
W2 relative error: 4.395471174488098e-08
W3 relative error: 6.244950801146958e-08
b1 relative error: 2.0413189453171298e-09
b2 relative error: 2.785132687644587e-09
b3 relative error: 1.3584809600954945e-10


Running check with dropout =  0.75
Initial loss:  2.30149284277
W1 relative error: 9.83241315204456e-08
W2 relative error: 1.2484064610109774e-07
W3 relative error: 7.072876331070812e-08
b1 relative error: 6.57071632012831e-08
b2 relative error: 2.4973364074125794e-09
b3 relative error: 1.5893216402745932e-10


Running check with dropout =  1.0
Initial loss:  2.30273025736
W1 relative error: 1.3610160941555372e-07
W2 relative error: 1.1681571878793191e-07
W3 relative error: 9.422106252896962e-08
b1 relative error: 1.1796170512665718e-08
b2 relative error: 2.062339792715741e-09
b3 relative error: 9.274709761196104e-11
```

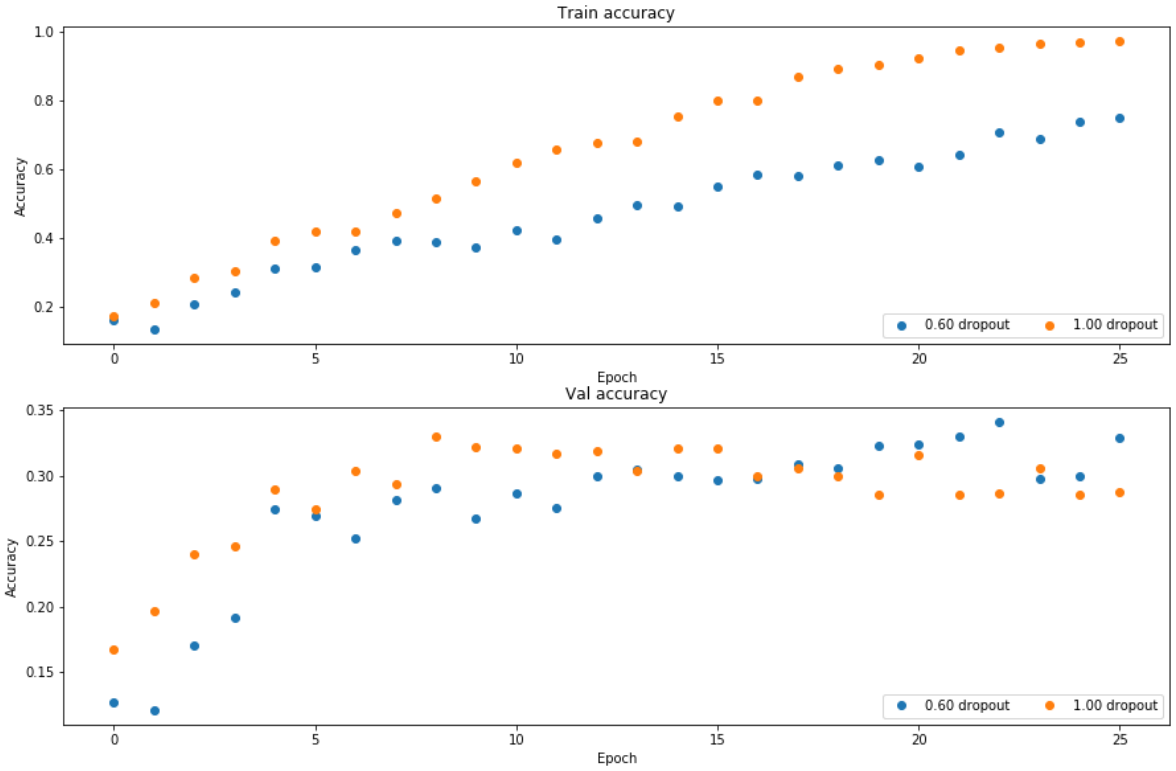
Dropout as a regularizer

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 [11]:
```
# 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: 2.300199
(Epoch 0 / 25) train acc: 0.158000; val_acc: 0.127000
(Epoch 1 / 25) train acc: 0.132000; val_acc: 0.121000
(Epoch 2 / 25) train acc: 0.204000; val_acc: 0.170000
(Epoch 3 / 25) train acc: 0.240000; val_acc: 0.192000
(Epoch 4 / 25) train acc: 0.312000; val_acc: 0.274000
(Epoch 5 / 25) train acc: 0.314000; val_acc: 0.269000
(Epoch 6 / 25) train acc: 0.364000; val_acc: 0.252000
(Epoch 7 / 25) train acc: 0.390000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.386000; val_acc: 0.290000
(Epoch 9 / 25) train acc: 0.372000; val_acc: 0.267000
(Epoch 10 / 25) train acc: 0.424000; val_acc: 0.286000
(Epoch 11 / 25) train acc: 0.396000; val_acc: 0.275000
(Epoch 12 / 25) train acc: 0.458000; val_acc: 0.299000
(Epoch 13 / 25) train acc: 0.496000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.492000; val_acc: 0.299000
(Epoch 15 / 25) train acc: 0.550000; val_acc: 0.296000
(Epoch 16 / 25) train acc: 0.584000; val_acc: 0.297000
(Epoch 17 / 25) train acc: 0.582000; val_acc: 0.309000
(Epoch 18 / 25) train acc: 0.612000; val_acc: 0.306000
(Epoch 19 / 25) train acc: 0.628000; val_acc: 0.323000
(Epoch 20 / 25) train acc: 0.608000; val_acc: 0.324000
(Iteration 101 / 125) loss: 1.369535
(Epoch 21 / 25) train acc: 0.644000; val_acc: 0.330000
(Epoch 22 / 25) train acc: 0.708000; val_acc: 0.341000
(Epoch 23 / 25) train acc: 0.690000; val_acc: 0.297000
(Epoch 24 / 25) train acc: 0.740000; val_acc: 0.300000
(Epoch 25 / 25) train acc: 0.750000; val_acc: 0.329000
(Iteration 1 / 125) loss: 2.300607
(Epoch 0 / 25) train acc: 0.172000; val_acc: 0.167000
(Epoch 1 / 25) train acc: 0.210000; val_acc: 0.197000
(Epoch 2 / 25) train acc: 0.284000; val_acc: 0.240000
(Epoch 3 / 25) train acc: 0.302000; val_acc: 0.246000
(Epoch 4 / 25) train acc: 0.392000; val_acc: 0.289000
(Epoch 5 / 25) train acc: 0.420000; val_acc: 0.274000
(Epoch 6 / 25) train acc: 0.420000; val_acc: 0.304000
(Epoch 7 / 25) train acc: 0.474000; val_acc: 0.293000
(Epoch 8 / 25) train acc: 0.516000; val_acc: 0.330000
(Epoch 9 / 25) train acc: 0.566000; val_acc: 0.322000
(Epoch 10 / 25) train acc: 0.620000; val_acc: 0.321000
(Epoch 11 / 25) train acc: 0.656000; val_acc: 0.317000
(Epoch 12 / 25) train acc: 0.676000; val_acc: 0.319000
(Epoch 13 / 25) train acc: 0.680000; val_acc: 0.304000
(Epoch 14 / 25) train acc: 0.754000; val_acc: 0.321000
(Epoch 15 / 25) train acc: 0.800000; val_acc: 0.321000
(Epoch 16 / 25) train acc: 0.800000; val_acc: 0.299000
(Epoch 17 / 25) train acc: 0.870000; val_acc: 0.306000
(Epoch 18 / 25) train acc: 0.892000; val_acc: 0.300000
(Epoch 19 / 25) train acc: 0.904000; val_acc: 0.285000
(Epoch 20 / 25) train acc: 0.924000; val_acc: 0.316000
(Iteration 101 / 125) loss: 0.255556
(Epoch 21 / 25) train acc: 0.948000; val_acc: 0.285000
(Epoch 22 / 25) train acc: 0.954000; val_acc: 0.286000
(Epoch 23 / 25) train acc: 0.966000; val_acc: 0.306000
(Epoch 24 / 25) train acc: 0.970000; val_acc: 0.285000
(Epoch 25 / 25) train acc: 0.974000; val_acc: 0.287000
```

In [12]:
```
# 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 drop
out' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
    plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropou
t' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

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



Question

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

Answer:

Yes. We can see that when we used dropout, the training accuracy was smaller than the accuracy of the model without dropout (which makes sense because the network with dropout is effectively with less neurons operating), but at the same time the validation accuracy remained roughly the same as the accuracy of the model that didn't incorporate dropout. Since the difference between the training and validation accuracy is bigger for the model with dropout, it means that dropout helped with the generalization of the model, which is a form of regularization.

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 [19]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Implement a FC-net that achieves at least 55% validation accuracy
#   on CIFAR-10.
# ================================================================ #
optimizer = 'adam'
layer_dims = [800, 650, 650]
weight_scale = 0.01
learning_rate = 1e-3
lr_decay = 0.9

model = FullyConnectedNet(layer_dims, weight_scale=weight_scale,
                          use_batchnorm=True, dropout=0.5)

solver = Solver(model, data,
                num_epochs=10, batch_size=150,
                update_rule=optimizer,
                optim_config={
                  'learning_rate': learning_rate,
                },
                lr_decay=lr_decay,
                verbose=True, print_every=50)
solver.train()

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

```
(Iteration 1 / 3260) loss: 2.309706
(Epoch 0 / 10) train acc: 0.197000; val_acc: 0.205000
(Iteration 51 / 3260) loss: 1.753303
(Iteration 101 / 3260) loss: 1.677952
(Iteration 151 / 3260) loss: 1.682492
(Iteration 201 / 3260) loss: 1.781695
(Iteration 251 / 3260) loss: 1.556207
(Iteration 301 / 3260) loss: 1.674324
(Epoch 1 / 10) train acc: 0.449000; val_acc: 0.471000
(Iteration 351 / 3260) loss: 1.666532
(Iteration 401 / 3260) loss: 1.574301
(Iteration 451 / 3260) loss: 1.462209
(Iteration 501 / 3260) loss: 1.506639
(Iteration 551 / 3260) loss: 1.467008
(Iteration 601 / 3260) loss: 1.592831
(Iteration 651 / 3260) loss: 1.517175
(Epoch 2 / 10) train acc: 0.522000; val_acc: 0.497000
(Iteration 701 / 3260) loss: 1.691948
(Iteration 751 / 3260) loss: 1.634745
(Iteration 801 / 3260) loss: 1.518882
(Iteration 851 / 3260) loss: 1.345086
(Iteration 901 / 3260) loss: 1.379469
(Iteration 951 / 3260) loss: 1.451329
(Epoch 3 / 10) train acc: 0.564000; val_acc: 0.507000
(Iteration 1001 / 3260) loss: 1.393822
(Iteration 1051 / 3260) loss: 1.323602
(Iteration 1101 / 3260) loss: 1.460633
(Iteration 1151 / 3260) loss: 1.457747
(Iteration 1201 / 3260) loss: 1.299865
(Iteration 1251 / 3260) loss: 1.294650
(Iteration 1301 / 3260) loss: 1.352543
(Epoch 4 / 10) train acc: 0.558000; val_acc: 0.526000
(Iteration 1351 / 3260) loss: 1.252901
(Iteration 1401 / 3260) loss: 1.276353
(Iteration 1451 / 3260) loss: 1.366664
(Iteration 1501 / 3260) loss: 1.298607
(Iteration 1551 / 3260) loss: 1.463464
(Iteration 1601 / 3260) loss: 1.429020
(Epoch 5 / 10) train acc: 0.538000; val_acc: 0.529000
(Iteration 1651 / 3260) loss: 1.417271
(Iteration 1701 / 3260) loss: 1.502363
(Iteration 1751 / 3260) loss: 1.194770
(Iteration 1801 / 3260) loss: 1.500889
(Iteration 1851 / 3260) loss: 1.297009
(Iteration 1901 / 3260) loss: 1.403091
(Iteration 1951 / 3260) loss: 1.318034
(Epoch 6 / 10) train acc: 0.582000; val_acc: 0.538000
(Iteration 2001 / 3260) loss: 1.275208
(Iteration 2051 / 3260) loss: 1.337344
(Iteration 2101 / 3260) loss: 1.348497
(Iteration 2151 / 3260) loss: 1.312881
(Iteration 2201 / 3260) loss: 1.283106
(Iteration 2251 / 3260) loss: 1.321507
(Epoch 7 / 10) train acc: 0.585000; val_acc: 0.535000
(Iteration 2301 / 3260) loss: 1.233513
(Iteration 2351 / 3260) loss: 1.255630
(Iteration 2401 / 3260) loss: 1.185922
```

```
(Iteration 2451 / 3260) loss: 1.326277
(Iteration 2501 / 3260) loss: 1.235982
(Iteration 2551 / 3260) loss: 1.212447
(Iteration 2601 / 3260) loss: 1.270914
(Epoch 8 / 10) train acc: 0.635000; val_acc: 0.555000
(Iteration 2651 / 3260) loss: 1.199987
(Iteration 2701 / 3260) loss: 1.346234
(Iteration 2751 / 3260) loss: 1.089702
(Iteration 2801 / 3260) loss: 1.153776
(Iteration 2851 / 3260) loss: 1.194224
(Iteration 2901 / 3260) loss: 1.269478
(Epoch 9 / 10) train acc: 0.609000; val_acc: 0.551000
(Iteration 2951 / 3260) loss: 1.030614
(Iteration 3001 / 3260) loss: 1.090293
(Iteration 3051 / 3260) loss: 1.232051
(Iteration 3101 / 3260) loss: 1.090456
(Iteration 3151 / 3260) loss: 1.244426
(Iteration 3201 / 3260) loss: 1.143223
(Iteration 3251 / 3260) loss: 1.317788
(Epoch 10 / 10) train acc: 0.629000; val_acc: 0.573000
```

4 Code for all sections

optim.py

```
import numpy as np

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

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

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

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

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

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

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


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

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

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


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

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

```
72
73      # ================================================================ #
74      # YOUR CODE HERE:
75      #   Implement the momentum update formula.  Return the updated weights
76      #   as next_w, and the updated velocity as v.
77      # ================================================================ #
78      v = config['momentum'] * v - config['learning_rate'] * dw
79      next_w = w + v
80      # ================================================================ #
81      # END YOUR CODE HERE
82      # ================================================================ #
83
84      config['velocity'] = v
85
86      return next_w, config
87
88  def sgd_nesterov_momentum(w, dw, config=None):
89      """
90      Performs stochastic gradient descent with Nesterov momentum.
91
92      config format:
93      - learning_rate: Scalar learning rate.
94      - momentum: Scalar between 0 and 1 giving the momentum value.
95        Setting momentum = 0 reduces to sgd.
96      - velocity: A numpy array of the same shape as w and dw used to store a moving
97        average of the gradients.
98      """
99      if config is None: config = {}
100     config.setdefault('learning_rate', 1e-2)
101     config.setdefault('momentum', 0.9) # set momentum to 0.9 if it wasn't there
102     v = config.get('velocity', np.zeros_like(w))   # gets velocity, else sets it to zero.
103
104     # ================================================================ #
105     # YOUR CODE HERE:
106     #   Implement the momentum update formula.  Return the updated weights
107     #   as next_w, and the updated velocity as v.
108     # ================================================================ #
109     v_old = v
110     v = config['momentum'] * v - config['learning_rate'] * dw
111     next_w = w + v + config['momentum'] * (v - v_old)
112     # ================================================================ #
113     # END YOUR CODE HERE
114     # ================================================================ #
115
116     config['velocity'] = v
117
118     return next_w, config
119
120 def rmsprop(w, dw, config=None):
121     """
122     Uses the RMSProp update rule, which uses a moving average of squared gradient
123     values to set adaptive per-parameter learning rates.
124
125     config format:
126     - learning_rate: Scalar learning rate.
127     - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
128       gradient cache.
129     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
130     - beta: Moving average of second moments of gradients.
131     """
132     if config is None: config = {}
133     config.setdefault('learning_rate', 1e-2)
134     config.setdefault('decay_rate', 0.99)
135     config.setdefault('epsilon', 1e-8)
136     config.setdefault('a', np.zeros_like(w))
137
138     next_w = None
139
140     # ================================================================ #
141     # YOUR CODE HERE:
142     #   Implement RMSProp.  Store the next value of w as next_w.  You need
143     #   to also store in config['a'] the moving average of the second
144     #   moment gradients, so they can be used for future gradients. Concretely,
145     #   config['a'] corresponds to "a" in the lecture notes.
```

```
    # ================================================================ #
    config['a'] = config['decay_rate'] * config['a'] + (1-config['decay_rate']) * np.multiply(dw, dw)
    next_w = w - np.multiply(config['learning_rate'] / (np.sqrt(config['a']) + config['epsilon']), dw)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config


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

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

    next_w = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement Adam.  Store the next value of w as next_w.  You need
    #   to also store in config['a'] the moving average of the second
    #   moment gradients, and in config['v'] the moving average of the
    #   first moments.  Finally, store in config['t'] the increasing time.
    # ================================================================ #
    config['t'] += 1
    config['v'] = config['beta1'] * config['v'] + (1-config['beta1']) * dw
    config['a'] = config['beta2'] * config['a'] + (1-config['beta2']) * np.multiply(dw, dw)
    v_corr = config['v']/(1-(config['beta1'] ** config['t']))
    a_corr = config['a']/(1-(config['beta2'] ** config['t']))
    next_w = w - np.multiply((config['learning_rate']/(np.sqrt(a_corr) + config['epsilon'])), v_corr)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return next_w, config

```

layers.py

```
import numpy as np
import pdb

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


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

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

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

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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the output of the forward pass.  Notice the dimensions
    #   of w are D x M, which is the transpose of what we did in earlier
    #   assignments.
    # ================================================================ #
    out = x.reshape(x.shape[0], -1).dot(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: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ================================================================ #

    # 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

    db = np.sum(dout, axis=0)
    dx = np.array(dout).dot(w.T).reshape(x.shape)
    dw = x.reshape(x.shape[0], -1).T.dot(dout)
    # ============================================================= #
    # 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 = x * (x > 0)
    # ============================================================= #
    # 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
    # ============================================================= #

    # ReLU directs linearly to those > 0
    dx = dout * (x > 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.
        # ============================================================ #

        running_mean = np.mean(x, axis=0)
        running_var = np.var(x, axis=0)
        x_hat = (x-running_mean) / np.sqrt(eps + running_var)
        out = gamma * x_hat + beta
        cache = (x_hat, x, running_mean, running_var, eps, gamma)

        # ============================================================ #
        # 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'.
        # ============================================================ #

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

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

```
229
230     return out, cache
231
232 def batchnorm_backward(dout, cache):
233     """
234     Backward pass for batch normalization.
235
236     For this implementation, you should write out a computation graph for
237     batch normalization on paper and propagate gradients backward through
238     intermediate nodes.
239
240     Inputs:
241     - dout: Upstream derivatives, of shape (N, D)
242     - cache: Variable of intermediates from batchnorm_forward.
243
244     Returns a tuple of:
245     - dx: Gradient with respect to inputs x, of shape (N, D)
246     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
247     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
248     """
249     dx, dgamma, dbeta = None, None, None
250
251     # ============================================================= #
252     # YOUR CODE HERE:
253     #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
254     # ============================================================= #
255     x_hat, x, mean, var, eps, gamma = cache
256     N, D = x.shape
257     dbeta = np.sum(dout, axis=0)
258     dgamma = np.sum(dout * x_hat, axis=0)
259     dl_dxhat = dout * gamma
260     dl_dvar = (-1/2) * np.sum((1/((var + eps) ** (3/2))) * (x - mean) * dl_dxhat, axis=0)
261     dl_dmean = (-1/(np.sqrt(var+eps))) * np.sum(dl_dxhat, axis=0)
262     dx = np.array((1/np.sqrt(var + eps)) * dl_dxhat + (2 * (x - mean) / N) * dl_dvar + (1/N) * dl_dmean)
263     # ============================================================= #
264     # END YOUR CODE HERE
265     # ============================================================= #
266
267     return dx, dgamma, dbeta
268
269 def dropout_forward(x, dropout_param):
270     """
271     Performs the forward pass for (inverted) dropout.
272
273     Inputs:
274     - x: Input data, of any shape
275     - dropout_param: A dictionary with the following keys:
276       - p: Dropout parameter. We keep each neuron output with probability p.
277       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
278         if the mode is test, then just return the input.
279       - seed: Seed for the random number generator. Passing seed makes this
280         function deterministic, which is needed for gradient checking but not in
281         real networks.
282
283     Outputs:
284     - out: Array of the same shape as x.
285     - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
286       mask that was used to multiply the input; in test mode, mask is None.
287     """
288     p, mode = dropout_param['p'], dropout_param['mode']
289     if 'seed' in dropout_param:
290         np.random.seed(dropout_param['seed'])
291
292     mask = None
293     out = None
294
295     if mode == 'train':
296         # ============================================================= #
297         # YOUR CODE HERE:
298         #   Implement the inverted dropout forward pass during training time.
299         #   Store the masked and scaled activations in out, and store the
300         #   dropout mask as the variable mask.
301         # ============================================================= #
302
303
304         mask = np.random.rand(*x.shape) < p
305         out = x * mask / p
```

```
306         # ================================================================ #
307         # END YOUR CODE HERE
308         # ================================================================ #
309 
310     elif mode == 'test':
311 
312         # ================================================================ #
313         # YOUR CODE HERE:
314         #   Implement the inverted dropout forward pass during test time.
315         # ================================================================ #
316 
317 
318         out = x
319         # ================================================================ #
320         # END YOUR CODE HERE
321         # ================================================================ #
322 
323     cache = (dropout_param, mask)
324     out = out.astype(x.dtype, copy=False)
325 
326     return out, cache
327 
328 def dropout_backward(dout, cache):
329     """
330     Perform the backward pass for (inverted) dropout.
331 
332     Inputs:
333     - dout: Upstream derivatives, of any shape
334     - cache: (dropout_param, mask) from dropout_forward.
335     """
336     dropout_param, mask = cache
337     mode = dropout_param['mode']
338 
339     dx = None
340     if mode == 'train':
341         # ================================================================ #
342         # YOUR CODE HERE:
343         #   Implement the inverted dropout backward pass during training time.
344         # ================================================================ #
345 
346 
347         dx = dout * mask / dropout_param['p']
348         # ================================================================ #
349         # END YOUR CODE HERE
350         # ================================================================ #
351     elif mode == 'test':
352         # ================================================================ #
353         # YOUR CODE HERE:
354         #   Implement the inverted dropout backward pass during test time.
355         # ================================================================ #
356 
357 
358         dx = dout
359         # ================================================================ #
360         # END YOUR CODE HERE
361         # ================================================================ #
362     return dx
363 
364 def svm_loss(x, y):
365     """
366     Computes the loss and gradient using for multiclass SVM classification.
367 
368     Inputs:
369     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
370       for the ith input.
371     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
372       0 <= y[i] < C
373 
374     Returns a tuple of:
375     - loss: Scalar giving the loss
376     - dx: Gradient of the loss with respect to x
377     """
378     N = x.shape[0]
379     correct_class_scores = x[np.arange(N), y]
380     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
381     margins[np.arange(N), y] = 0
382     loss = np.sum(margins) / N
```

```
383     num_pos = np.sum(margins > 0, axis=1)
384     dx = np.zeros_like(x)
385     dx[margins > 0] = 1
386     dx[np.arange(N), y] -= num_pos
387     dx /= N
388     return loss, dx
389 
390 
391 def softmax_loss(x, y):
392     """
393     Computes the loss and gradient for softmax classification.
394 
395     Inputs:
396     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
397       for the ith input.
398     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
399       0 <= y[i] < C
400 
401     Returns a tuple of:
402     - loss: Scalar giving the loss
403     - dx: Gradient of the loss with respect to x
404     """
405 
406     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
407     probs /= np.sum(probs, axis=1, keepdims=True)
408     N = x.shape[0]
409     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
410     dx = probs.copy()
411     dx[np.arange(N), y] -= 1
412     dx /= N
413     return loss, dx
414 
```

layer_utils.py

```python
from .layers import *

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

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

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

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


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

def affine_batchnorm_relu_forward(x, w, b, gamma, beta, bn_param):
  """
  Convenience layer that performs an affine transform followed by batch normalization and then ReLU

  Inputs:
  - x: Input to the affine layer
  - w, b: Weights for the affine layer
  - gamma, beta: the gamma and beta parameters associated with this layer.
  - bn_param: a set of parameters corresponding to the layer. Relevant mainly for testing

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

def affine_batchnorm_relu_backward(dout, cache):
  """
  Backward pass for the affine-batchnorm-relu convenience layer
  """
  fc_cache, bn_cache, relu_cache = cache
  da = relu_backward(dout, relu_cache)
  da, dgamma, dbeta = batchnorm_backward(da, bn_cache)
  dx, dw, db = affine_backward(da, fc_cache)
  return dx, dw, db, dgamma, dbeta

```

fc_net.py

```
1   import numpy as np
2   import pdb
3
4   from .layers import *
5   from .layer_utils import *
6
7   """
8   This code was originally written for CS 231n at Stanford University
9   (cs231n.stanford.edu).  It has been modified in various areas for use in the
10  ECE 239AS class at UCLA.  This includes the descriptions of what code to
11  implement as well as some slight potential changes in variable names to be
12  consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
13  permission to use this code.  To see the original version, please visit
14  cs231n.stanford.edu.
15  """
16
17  class FullyConnectedNet(object):
18      """
19      A fully-connected neural network with an arbitrary number of hidden layers,
20      ReLU nonlinearities, and a softmax loss function. This will also implement
21      dropout and batch normalization as options. For a network with L layers,
22      the architecture will be
23
24      {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax
25
26      where batch normalization and dropout are optional, and the {...} block is
27      repeated L - 1 times.
28
29      Similar to the TwoLayerNet above, learnable parameters are stored in the
30      self.params dictionary and will be learned using the Solver class.
31      """
32
33      def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
34                   dropout=0, use_batchnorm=False, reg=0.0,
35                   weight_scale=1e-2, dtype=np.float32, seed=None):
36          """
37          Initialize a new FullyConnectedNet.
38
39          Inputs:
40          - hidden_dims: A list of integers giving the size of each hidden layer.
41          - input_dim: An integer giving the size of the input.
42          - num_classes: An integer giving the number of classes to classify.
43          - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=1 then
44            the network should not use dropout at all.
45          - use_batchnorm: Whether or not the network should use batch normalization.
46          - reg: Scalar giving L2 regularization strength.
47          - weight_scale: Scalar giving the standard deviation for random
48            initialization of the weights.
49          - dtype: A numpy datatype object; all computations will be performed using
50            this datatype. float32 is faster but less accurate, so you should use
51            float64 for numeric gradient checking.
52          - seed: If not None, then pass this random seed to the dropout layers. This
53            will make the dropout layers deteriminstic so we can gradient check the
54            model.
55          """
56          self.use_batchnorm = use_batchnorm
57          self.use_dropout = dropout < 1
58          self.reg = reg
59          self.num_layers = 1 + len(hidden_dims)
60          self.dtype = dtype
61          self.params = {}
62
63          # ============================================================ #
64          # YOUR CODE HERE:
65          #   Initialize all parameters of the network in the self.params dictionary.
66          #   The weights and biases of layer 1 are W1 and b1; and in general the
67          #   weights and biases of layer i are Wi and bi. The
68          #   biases are initialized to zero and the weights are initialized
69          #   so that each parameter has mean 0 and standard deviation weight_scale.
70          #
71          #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
72          #   parameters to zero.  The gamma and beta parameters for layer 1 should
73          #   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
74          #   should be gamma2 and beta2, etc. Only use batchnorm if self.use_batchnorm
75          #   is true and DO NOT do batch normalize the output scores.
76          # ============================================================ #
77
78          next_layer_input_dim = input_dim
79          for i, hidden_dim in enumerate(hidden_dims, start=1):
80              self.params['W' + str(i)] = weight_scale * np.random.randn(next_layer_input_dim, hidden_dim)
```

```
81              self.params['b' + str(i)] = np.zeros(hidden_dim)
82              if self.use_batchnorm:
83                  self.params['gamma' + str(i)] = np.ones(hidden_dim)
84                  self.params['beta' + str(i)] = np.zeros(hidden_dim)
85              next_layer_input_dim = hidden_dim
86
87          self.params['W' + str(self.num_layers)] = weight_scale * np.random.randn(next_layer_input_dim, num_classes)
88          self.params['b' + str(self.num_layers)] = np.zeros(num_classes)
89
90          # ============================================================ #
91          # END YOUR CODE HERE
92          # ============================================================ #
93
94          # When using dropout we need to pass a dropout_param dictionary to each
95          # dropout layer so that the layer knows the dropout probability and the mode
96          # (train / test). You can pass the same dropout_param to each dropout layer.
97          self.dropout_param = {}
98          if self.use_dropout:
99              self.dropout_param = {'mode': 'train', 'p': dropout}
100         if seed is not None:
101             self.dropout_param['seed'] = seed
102
103         # With batch normalization we need to keep track of running means and
104         # variances, so we need to pass a special bn_param object to each batch
105         # normalization layer. You should pass self.bn_params[0] to the forward pass
106         # of the first batch normalization layer, self.bn_params[1] to the forward
107         # pass of the second batch normalization layer, etc.
108         self.bn_params = []
109         if self.use_batchnorm:
110             self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
111
112         # Cast all parameters to the correct datatype
113         for k, v in self.params.items():
114             self.params[k] = v.astype(dtype)
115
116
117     def loss(self, X, y=None):
118         """
119         Compute loss and gradient for the fully-connected net.
120
121         Input / output: Same as TwoLayerNet above.
122         """
123         X = X.astype(self.dtype)
124         mode = 'test' if y is None else 'train'
125
126         # Set train/test mode for batchnorm params and dropout param since they
127         # behave differently during training and testing.
128         if self.dropout_param is not None:
129             self.dropout_param['mode'] = mode
130         if self.use_batchnorm:
131             for bn_param in self.bn_params:
132                 bn_param[mode] = mode
133
134         scores = None
135
136         # ============================================================ #
137         # YOUR CODE HERE:
138         #   Implement the forward pass of the FC net and store the output
139         #   scores as the variable "scores".
140         #
141         #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
142         #   between the affine_forward and relu_forward layers.  You may
143         #   also write an affine_batchnorm_relu() function in layer_utils.py.
144         #
145         #   DROPOUT: If dropout is non-zero, insert a dropout layer after
146         #   every ReLU layer.
147         # ============================================================ #
148
149         caches = {}
150         dropout_caches = {}
151         layer_out = X
152         for i in range(1, self.num_layers):
153             if self.use_batchnorm:
154                 layer_out, caches[i] = affine_batchnorm_relu_forward(np.array(layer_out),
155                                                      self.params['W' + str(i)],
156                                                      self.params['b' + str(i)],
157                                                      self.params['gamma' + str(i)],
158                                                      self.params['beta' + str(i)],
159                                                      self.bn_params[i - 1])
160             else: # No batchnorm
161                 layer_out, caches[i] = affine_relu_forward(np.array(layer_out),
162                                                    self.params['W' + str(i)],
163                                                    self.params['b' + str(i)])
```

```
            if self.use_dropout:
                layer_out, dropout_caches[i] = dropout_forward(layer_out, self.dropout_param)



        scores, caches[self.num_layers] = affine_forward(np.array(layer_out),
                                                         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.
        # ============================================================ #

        sum_of_W_norms = 0
        for i in range(self.num_layers):
            sum_of_W_norms += (np.linalg.norm(self.params['W' + str(i + 1)]) ** 2)

        loss, soft_grad = softmax_loss(scores, y)
        loss += 0.5 * self.reg * sum_of_W_norms

        layer_back_grad, grads['W' + str(self.num_layers)], grads['b' + str(self.num_layers)] = \
            affine_backward(soft_grad, caches[self.num_layers])
        grads['W' + str(self.num_layers)] += self.reg * self.params['W' + str(self.num_layers)]

        for i in range(self.num_layers - 1, 0, -1):
            if self.use_dropout:
                layer_back_grad = dropout_backward(layer_back_grad, dropout_caches[i])
            if self.use_batchnorm:
                layer_back_grad, grads['W' + str(i)], grads['b' + str(i)], \
                grads['gamma' + str(i)], grads['beta' + str(i)] = \
                    affine_batchnorm_relu_backward(layer_back_grad, caches[i])
            else:
                layer_back_grad, grads['W' + str(i)], grads['b' + str(i)] = affine_relu_backward(layer_back_grad,
                                                                                                 caches[i])
            grads['W' + str(i)] += self.reg * self.params['W' + str(i)]

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

        return loss, grads

```