Fully-Connected Neural Nets

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

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

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

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

Acknowledgement: This exercise is adapted from Stanford CS231n.

In [12]:

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.fc_net import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from libs.solver import Solver

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

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

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

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

In [13]:

```
# Load the (preprocessed) CIFAR10 data.

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

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

Affine layer: foward

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

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

In [21]:

```
# Test the affine_forward function

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

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

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

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

In [25]:

```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

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

In [27]:

```
# Test the relu_forward function
```

```
x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

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

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

ReLU activation: backward

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

In [30]:

```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

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

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

"Sandwich" layers

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

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

In [31]:

```
from libs.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, dout)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax

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

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

In [32]:

```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

Two-layer network

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

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

In [35]:

```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
```

```python
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

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

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

In [50]:

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

model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

dataset = {
      'X_train': data["X_train"], # training data
      'y_train': data["y_train"], # training labels
      'X_val': data["X_val"], # validation data
      'y_val': data["y_val"] # validation labels
    }

solver = Solver(model, dataset, optim_config={'learning_rate': 1e-3})
solver.train()

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

```
(Iteration 1 / 4900) loss: 2.306162
(Epoch 0 / 10) train acc: 0.120000; val_acc: 0.113000
(Iteration 11 / 4900) loss: 2.249016
(Iteration 21 / 4900) loss: 2.165029
(Iteration 31 / 4900) loss: 2.067593
(Iteration 41 / 4900) loss: 2.025559
(Iteration 51 / 4900) loss: 2.152806
(Iteration 61 / 4900) loss: 1.904491
(Iteration 71 / 4900) loss: 1.845887
(Iteration 81 / 4900) loss: 1.670887
(Iteration 91 / 4900) loss: 1.853978
(Iteration 101 / 4900) loss: 1.785750
```

```
(Iteration 111 / 4900) loss: 1.941374
(Iteration 121 / 4900) loss: 1.876342
(Iteration 131 / 4900) loss: 1.889186
(Iteration 141 / 4900) loss: 1.675717
(Iteration 151 / 4900) loss: 1.857227
(Iteration 161 / 4900) loss: 1.647874
(Iteration 171 / 4900) loss: 1.699232
(Iteration 181 / 4900) loss: 1.636844
(Iteration 191 / 4900) loss: 1.649504
(Iteration 201 / 4900) loss: 1.674168
(Iteration 211 / 4900) loss: 1.651592
(Iteration 221 / 4900) loss: 1.635083
(Iteration 231 / 4900) loss: 1.807900
(Iteration 241 / 4900) loss: 1.828225
(Iteration 251 / 4900) loss: 1.617601
(Iteration 261 / 4900) loss: 1.890491
(Iteration 271 / 4900) loss: 1.728749
(Iteration 281 / 4900) loss: 1.575229
(Iteration 291 / 4900) loss: 1.677051
(Iteration 301 / 4900) loss: 1.606943
(Iteration 311 / 4900) loss: 1.576856
(Iteration 321 / 4900) loss: 1.618816
(Iteration 331 / 4900) loss: 1.532362
(Iteration 341 / 4900) loss: 1.414092
(Iteration 351 / 4900) loss: 1.701993
(Iteration 361 / 4900) loss: 1.441317
(Iteration 371 / 4900) loss: 1.475303
(Iteration 381 / 4900) loss: 1.493042
(Iteration 391 / 4900) loss: 1.871308
(Iteration 401 / 4900) loss: 1.536097
(Iteration 411 / 4900) loss: 1.424652
(Iteration 421 / 4900) loss: 1.661716
(Iteration 431 / 4900) loss: 1.668528
(Iteration 441 / 4900) loss: 1.520678
(Iteration 451 / 4900) loss: 1.557366
(Iteration 461 / 4900) loss: 1.443712
(Iteration 471 / 4900) loss: 1.489487
(Iteration 481 / 4900) loss: 1.611174
(Epoch 1 / 10) train acc: 0.480000; val_acc: 0.457000
(Iteration 491 / 4900) loss: 1.475236
(Iteration 501 / 4900) loss: 1.743354
(Iteration 511 / 4900) loss: 1.484650
(Iteration 521 / 4900) loss: 1.305192
(Iteration 531 / 4900) loss: 1.508060
(Iteration 541 / 4900) loss: 1.555800
(Iteration 551 / 4900) loss: 1.369424
(Iteration 561 / 4900) loss: 1.580629
(Iteration 571 / 4900) loss: 1.613584
(Iteration 581 / 4900) loss: 1.509086
(Iteration 591 / 4900) loss: 1.588791
(Iteration 601 / 4900) loss: 1.602585
(Iteration 611 / 4900) loss: 1.599155
(Iteration 621 / 4900) loss: 1.568520
(Iteration 631 / 4900) loss: 1.338827
(Iteration 641 / 4900) loss: 1.462608
(Iteration 651 / 4900) loss: 1.749024
(Iteration 661 / 4900) loss: 1.594485
(Iteration 671 / 4900) loss: 1.449679
(Iteration 681 / 4900) loss: 1.599995
(Iteration 691 / 4900) loss: 1.660691
(Iteration 701 / 4900) loss: 1.716781
(Iteration 711 / 4900) loss: 1.420303
(Iteration 721 / 4900) loss: 1.528623
(Iteration 731 / 4900) loss: 1.666009
(Iteration 741 / 4900) loss: 1.320328
(Iteration 751 / 4900) loss: 1.516522
(Iteration 761 / 4900) loss: 1.581535
(Iteration 771 / 4900) loss: 1.323565
(Iteration 781 / 4900) loss: 1.496120
(Iteration 791 / 4900) loss: 1.333479
(Iteration 801 / 4900) loss: 1.297135
(Iteration 811 / 4900) loss: 1.649749
(Iteration 821 / 4900) loss: 1.450752
(Iteration 831 / 4900) loss: 1.537832
(Iteration 841 / 4900) loss: 1.385218
(Iteration 851 / 4900) loss: 1.520944
(Iteration 861 / 4900) loss: 1.721423
(Iteration 871 / 4900) loss: 1.523786
(Iteration 881 / 4900) loss: 1.346369
(Iteration 891 / 4900) loss: 1.563386
(Iteration 901 / 4900) loss: 1.473082
(Iteration 911 / 4900) loss: 1.382064
(Iteration 921 / 4900) loss: 1.620143
(Iteration 931 / 4900) loss: 1.364246
(Iteration 941 / 4900) loss: 1.295611
(Iteration 951 / 4900) loss: 1.276734
(Iteration 961 / 4900) loss: 1.590876
(Iteration 971 / 4900) loss: 1.495068
(Epoch 2 / 10) train acc: 0.461000; val_acc: 0.464000
(Iteration 981 / 4900) loss: 1.277175
(Iteration 991 / 4900) loss: 1.413304
(Iteration 1001 / 4900) loss: 1.563123
(Iteration 1011 / 4900) loss: 1.408717
(Iteration 1021 / 4900) loss: 1.104940
```

```
(Iteration 1021 / 4900) loss: 1.104940
(Iteration 1031 / 4900) loss: 1.526906
(Iteration 1041 / 4900) loss: 1.488850
(Iteration 1051 / 4900) loss: 1.472523
(Iteration 1061 / 4900) loss: 1.653504
(Iteration 1071 / 4900) loss: 1.407780
(Iteration 1081 / 4900) loss: 1.407828
(Iteration 1091 / 4900) loss: 1.407177
(Iteration 1101 / 4900) loss: 1.674800
(Iteration 1111 / 4900) loss: 1.542447
(Iteration 1121 / 4900) loss: 1.329605
(Iteration 1131 / 4900) loss: 1.438559
(Iteration 1141 / 4900) loss: 1.388235
(Iteration 1151 / 4900) loss: 1.381584
(Iteration 1161 / 4900) loss: 1.305406
(Iteration 1171 / 4900) loss: 1.485775
(Iteration 1181 / 4900) loss: 1.613597
(Iteration 1191 / 4900) loss: 1.378534
(Iteration 1201 / 4900) loss: 1.289464
(Iteration 1211 / 4900) loss: 1.347027
(Iteration 1221 / 4900) loss: 1.294200
(Iteration 1231 / 4900) loss: 1.344029
(Iteration 1241 / 4900) loss: 1.336155
(Iteration 1251 / 4900) loss: 1.592745
(Iteration 1261 / 4900) loss: 1.396798
(Iteration 1271 / 4900) loss: 1.389956
(Iteration 1281 / 4900) loss: 1.594764
(Iteration 1291 / 4900) loss: 1.308080
(Iteration 1301 / 4900) loss: 1.330416
(Iteration 1311 / 4900) loss: 1.349715
(Iteration 1321 / 4900) loss: 1.240841
(Iteration 1331 / 4900) loss: 1.157964
(Iteration 1341 / 4900) loss: 1.269385
(Iteration 1351 / 4900) loss: 1.445325
(Iteration 1361 / 4900) loss: 1.470050
(Iteration 1371 / 4900) loss: 1.346539
(Iteration 1381 / 4900) loss: 1.187804
(Iteration 1391 / 4900) loss: 1.481951
(Iteration 1401 / 4900) loss: 1.417730
(Iteration 1411 / 4900) loss: 1.362582
(Iteration 1421 / 4900) loss: 1.502524
(Iteration 1431 / 4900) loss: 1.417385
(Iteration 1441 / 4900) loss: 1.536422
(Iteration 1451 / 4900) loss: 1.294020
(Iteration 1461 / 4900) loss: 1.363310
(Epoch 3 / 10) train acc: 0.487000; val_acc: 0.474000
(Iteration 1471 / 4900) loss: 1.410159
(Iteration 1481 / 4900) loss: 1.310651
(Iteration 1491 / 4900) loss: 1.388494
(Iteration 1501 / 4900) loss: 1.070457
(Iteration 1511 / 4900) loss: 1.180705
(Iteration 1521 / 4900) loss: 1.343998
(Iteration 1531 / 4900) loss: 1.348084
(Iteration 1541 / 4900) loss: 1.374491
(Iteration 1551 / 4900) loss: 1.327563
(Iteration 1561 / 4900) loss: 1.254932
(Iteration 1571 / 4900) loss: 1.620541
(Iteration 1581 / 4900) loss: 1.267205
(Iteration 1591 / 4900) loss: 1.262163
(Iteration 1601 / 4900) loss: 1.240588
(Iteration 1611 / 4900) loss: 1.390065
(Iteration 1621 / 4900) loss: 1.175629
(Iteration 1631 / 4900) loss: 1.433537
(Iteration 1641 / 4900) loss: 1.489298
(Iteration 1651 / 4900) loss: 1.328493
(Iteration 1661 / 4900) loss: 1.523000
(Iteration 1671 / 4900) loss: 1.595195
(Iteration 1681 / 4900) loss: 1.286864
(Iteration 1691 / 4900) loss: 1.334336
(Iteration 1701 / 4900) loss: 1.325838
(Iteration 1711 / 4900) loss: 1.340235
(Iteration 1721 / 4900) loss: 1.375396
(Iteration 1731 / 4900) loss: 1.442346
(Iteration 1741 / 4900) loss: 1.420037
(Iteration 1751 / 4900) loss: 1.242938
(Iteration 1761 / 4900) loss: 1.401430
(Iteration 1771 / 4900) loss: 1.298148
(Iteration 1781 / 4900) loss: 1.493461
(Iteration 1791 / 4900) loss: 1.487496
(Iteration 1801 / 4900) loss: 1.314425
(Iteration 1811 / 4900) loss: 1.471182
(Iteration 1821 / 4900) loss: 1.232841
(Iteration 1831 / 4900) loss: 1.594184
(Iteration 1841 / 4900) loss: 1.683545
(Iteration 1851 / 4900) loss: 1.363586
(Iteration 1861 / 4900) loss: 1.282471
(Iteration 1871 / 4900) loss: 1.184138
(Iteration 1881 / 4900) loss: 1.262712
(Iteration 1891 / 4900) loss: 1.428266
(Iteration 1901 / 4900) loss: 1.522882
(Iteration 1911 / 4900) loss: 1.387742
(Iteration 1921 / 4900) loss: 1.411500
(Iteration 1931 / 4900) loss: 1.152679
(Iteration 1941 / 4900) loss: 1.499471
```

```
(Iteration 1951 / 4900) loss: 1.149826
(Epoch 4 / 10) train acc: 0.547000; val_acc: 0.488000
(Iteration 1961 / 4900) loss: 1.286245
(Iteration 1971 / 4900) loss: 1.482067
(Iteration 1981 / 4900) loss: 1.140811
(Iteration 1991 / 4900) loss: 1.331041
(Iteration 2001 / 4900) loss: 1.606888
(Iteration 2011 / 4900) loss: 1.432029
(Iteration 2021 / 4900) loss: 1.302358
(Iteration 2031 / 4900) loss: 1.390731
(Iteration 2041 / 4900) loss: 1.574133
(Iteration 2051 / 4900) loss: 1.262312
(Iteration 2061 / 4900) loss: 1.360025
(Iteration 2071 / 4900) loss: 1.315767
(Iteration 2081 / 4900) loss: 1.362490
(Iteration 2091 / 4900) loss: 1.450390
(Iteration 2101 / 4900) loss: 1.387158
(Iteration 2111 / 4900) loss: 1.201113
(Iteration 2121 / 4900) loss: 1.246484
(Iteration 2131 / 4900) loss: 1.416665
(Iteration 2141 / 4900) loss: 1.287599
(Iteration 2151 / 4900) loss: 1.356543
(Iteration 2161 / 4900) loss: 1.507685
(Iteration 2171 / 4900) loss: 1.162599
(Iteration 2181 / 4900) loss: 1.364510
(Iteration 2191 / 4900) loss: 1.283744
(Iteration 2201 / 4900) loss: 1.234146
(Iteration 2211 / 4900) loss: 1.402752
(Iteration 2221 / 4900) loss: 1.285512
(Iteration 2231 / 4900) loss: 1.310993
(Iteration 2241 / 4900) loss: 1.500633
(Iteration 2251 / 4900) loss: 1.539763
(Iteration 2261 / 4900) loss: 1.308142
(Iteration 2271 / 4900) loss: 1.111819
(Iteration 2281 / 4900) loss: 1.187900
(Iteration 2291 / 4900) loss: 1.322684
(Iteration 2301 / 4900) loss: 1.444188
(Iteration 2311 / 4900) loss: 1.093848
(Iteration 2321 / 4900) loss: 1.271365
(Iteration 2331 / 4900) loss: 1.189386
(Iteration 2341 / 4900) loss: 1.442172
(Iteration 2351 / 4900) loss: 1.249779
(Iteration 2361 / 4900) loss: 1.339812
(Iteration 2371 / 4900) loss: 1.295459
(Iteration 2381 / 4900) loss: 1.388452
(Iteration 2391 / 4900) loss: 1.352695
(Iteration 2401 / 4900) loss: 1.141493
(Iteration 2411 / 4900) loss: 1.334437
(Iteration 2421 / 4900) loss: 1.209058
(Iteration 2431 / 4900) loss: 1.330664
(Iteration 2441 / 4900) loss: 1.299373
(Epoch 5 / 10) train acc: 0.560000; val_acc: 0.461000
(Iteration 2451 / 4900) loss: 1.284254
(Iteration 2461 / 4900) loss: 1.391360
(Iteration 2471 / 4900) loss: 1.158865
(Iteration 2481 / 4900) loss: 1.358796
(Iteration 2491 / 4900) loss: 1.250046
(Iteration 2501 / 4900) loss: 1.316442
(Iteration 2511 / 4900) loss: 1.243563
(Iteration 2521 / 4900) loss: 1.450900
(Iteration 2531 / 4900) loss: 1.331598
(Iteration 2541 / 4900) loss: 1.317036
(Iteration 2551 / 4900) loss: 1.283078
(Iteration 2561 / 4900) loss: 1.288864
(Iteration 2571 / 4900) loss: 1.457467
(Iteration 2581 / 4900) loss: 1.392945
(Iteration 2591 / 4900) loss: 1.024602
(Iteration 2601 / 4900) loss: 1.140737
(Iteration 2611 / 4900) loss: 1.390707
(Iteration 2621 / 4900) loss: 1.263738
(Iteration 2631 / 4900) loss: 1.195075
(Iteration 2641 / 4900) loss: 1.481943
(Iteration 2651 / 4900) loss: 1.425853
(Iteration 2661 / 4900) loss: 1.458809
(Iteration 2671 / 4900) loss: 1.337100
(Iteration 2681 / 4900) loss: 1.332990
(Iteration 2691 / 4900) loss: 1.379244
(Iteration 2701 / 4900) loss: 1.409523
(Iteration 2711 / 4900) loss: 1.190928
(Iteration 2721 / 4900) loss: 1.105199
(Iteration 2731 / 4900) loss: 1.472675
(Iteration 2741 / 4900) loss: 1.288850
(Iteration 2751 / 4900) loss: 1.431437
(Iteration 2761 / 4900) loss: 1.246488
(Iteration 2771 / 4900) loss: 1.164165
(Iteration 2781 / 4900) loss: 1.407029
(Iteration 2791 / 4900) loss: 1.339638
(Iteration 2801 / 4900) loss: 1.155990
(Iteration 2811 / 4900) loss: 1.206240
(Iteration 2821 / 4900) loss: 1.525694
(Iteration 2831 / 4900) loss: 1.194768
(Iteration 2841 / 4900) loss: 1.303345
(Iteration 2851 / 4900) loss: 1.362740
```

```
(Iteration 2861 / 4900) loss: 1.251342
(Iteration 2871 / 4900) loss: 1.419493
(Iteration 2881 / 4900) loss: 1.428141
(Iteration 2891 / 4900) loss: 1.171147
(Iteration 2901 / 4900) loss: 1.222072
(Iteration 2911 / 4900) loss: 1.503950
(Iteration 2921 / 4900) loss: 1.226686
(Iteration 2931 / 4900) loss: 1.208153
(Epoch 6 / 10) train acc: 0.537000; val_acc: 0.501000
(Iteration 2941 / 4900) loss: 1.241038
(Iteration 2951 / 4900) loss: 1.292549
(Iteration 2961 / 4900) loss: 1.230423
(Iteration 2971 / 4900) loss: 1.169870
(Iteration 2981 / 4900) loss: 1.535360
(Iteration 2991 / 4900) loss: 1.418277
(Iteration 3001 / 4900) loss: 1.161951
(Iteration 3011 / 4900) loss: 1.560548
(Iteration 3021 / 4900) loss: 1.119779
(Iteration 3031 / 4900) loss: 1.334325
(Iteration 3041 / 4900) loss: 1.212393
(Iteration 3051 / 4900) loss: 1.208114
(Iteration 3061 / 4900) loss: 1.239512
(Iteration 3071 / 4900) loss: 1.394738
(Iteration 3081 / 4900) loss: 1.049883
(Iteration 3091 / 4900) loss: 1.288109
(Iteration 3101 / 4900) loss: 1.316610
(Iteration 3111 / 4900) loss: 1.427603
(Iteration 3121 / 4900) loss: 1.200435
(Iteration 3131 / 4900) loss: 1.398025
(Iteration 3141 / 4900) loss: 1.085309
(Iteration 3151 / 4900) loss: 1.101389
(Iteration 3161 / 4900) loss: 1.234213
(Iteration 3171 / 4900) loss: 1.291455
(Iteration 3181 / 4900) loss: 1.254579
(Iteration 3191 / 4900) loss: 1.376855
(Iteration 3201 / 4900) loss: 1.299678
(Iteration 3211 / 4900) loss: 1.105246
(Iteration 3221 / 4900) loss: 1.202094
(Iteration 3231 / 4900) loss: 1.186104
(Iteration 3241 / 4900) loss: 1.171943
(Iteration 3251 / 4900) loss: 1.066226
(Iteration 3261 / 4900) loss: 1.354858
(Iteration 3271 / 4900) loss: 1.270477
(Iteration 3281 / 4900) loss: 1.267803
(Iteration 3291 / 4900) loss: 1.194475
(Iteration 3301 / 4900) loss: 1.179008
(Iteration 3311 / 4900) loss: 1.162406
(Iteration 3321 / 4900) loss: 1.232731
(Iteration 3331 / 4900) loss: 1.172218
(Iteration 3341 / 4900) loss: 1.320951
(Iteration 3351 / 4900) loss: 1.179220
(Iteration 3361 / 4900) loss: 1.400155
(Iteration 3371 / 4900) loss: 1.139927
(Iteration 3381 / 4900) loss: 1.320186
(Iteration 3391 / 4900) loss: 1.287997
(Iteration 3401 / 4900) loss: 1.215376
(Iteration 3411 / 4900) loss: 1.267104
(Iteration 3421 / 4900) loss: 1.332574
(Epoch 7 / 10) train acc: 0.564000; val_acc: 0.486000
(Iteration 3431 / 4900) loss: 1.305187
(Iteration 3441 / 4900) loss: 1.230709
(Iteration 3451 / 4900) loss: 1.289497
(Iteration 3461 / 4900) loss: 0.976482
(Iteration 3471 / 4900) loss: 1.287851
(Iteration 3481 / 4900) loss: 1.377976
(Iteration 3491 / 4900) loss: 1.245786
(Iteration 3501 / 4900) loss: 1.382857
(Iteration 3511 / 4900) loss: 1.351422
(Iteration 3521 / 4900) loss: 1.348351
(Iteration 3531 / 4900) loss: 1.326715
(Iteration 3541 / 4900) loss: 1.203300
(Iteration 3551 / 4900) loss: 1.410687
(Iteration 3561 / 4900) loss: 1.519472
(Iteration 3571 / 4900) loss: 1.408774
(Iteration 3581 / 4900) loss: 1.240295
(Iteration 3591 / 4900) loss: 1.128398
(Iteration 3601 / 4900) loss: 1.213940
(Iteration 3611 / 4900) loss: 1.087492
(Iteration 3621 / 4900) loss: 1.402258
(Iteration 3631 / 4900) loss: 1.114385
(Iteration 3641 / 4900) loss: 1.394379
(Iteration 3651 / 4900) loss: 1.276989
(Iteration 3661 / 4900) loss: 1.115998
(Iteration 3671 / 4900) loss: 1.334339
(Iteration 3681 / 4900) loss: 1.343234
(Iteration 3691 / 4900) loss: 1.282183
(Iteration 3701 / 4900) loss: 1.092918
(Iteration 3711 / 4900) loss: 1.432280
(Iteration 3721 / 4900) loss: 1.292198
(Iteration 3731 / 4900) loss: 1.189351
(Iteration 3741 / 4900) loss: 1.287728
(Iteration 3751 / 4900) loss: 1.310009
(Iteration 3761 / 4900) loss: 1.208918
(Iteration 3771 / 4900) loss: 1.339549
```

```
(Iteration 3781 / 4900) loss: 1.190091
(Iteration 3791 / 4900) loss: 1.418020
(Iteration 3801 / 4900) loss: 1.044485
(Iteration 3811 / 4900) loss: 1.353431
(Iteration 3821 / 4900) loss: 1.290448
(Iteration 3831 / 4900) loss: 1.240679
(Iteration 3841 / 4900) loss: 1.203523
(Iteration 3851 / 4900) loss: 1.335015
(Iteration 3861 / 4900) loss: 1.384638
(Iteration 3871 / 4900) loss: 1.269798
(Iteration 3881 / 4900) loss: 1.204511
(Iteration 3891 / 4900) loss: 1.334732
(Iteration 3901 / 4900) loss: 1.161617
(Iteration 3911 / 4900) loss: 1.203461
(Epoch 8 / 10) train acc: 0.575000; val_acc: 0.494000
(Iteration 3921 / 4900) loss: 1.225215
(Iteration 3931 / 4900) loss: 1.202740
(Iteration 3941 / 4900) loss: 1.191642
(Iteration 3951 / 4900) loss: 1.061411
(Iteration 3961 / 4900) loss: 1.308271
(Iteration 3971 / 4900) loss: 1.447250
(Iteration 3981 / 4900) loss: 1.232737
(Iteration 3991 / 4900) loss: 1.421253
(Iteration 4001 / 4900) loss: 1.195129
(Iteration 4011 / 4900) loss: 1.419883
(Iteration 4021 / 4900) loss: 1.377124
(Iteration 4031 / 4900) loss: 1.324195
(Iteration 4041 / 4900) loss: 1.217345
(Iteration 4051 / 4900) loss: 0.950084
(Iteration 4061 / 4900) loss: 1.189249
(Iteration 4071 / 4900) loss: 1.332915
(Iteration 4081 / 4900) loss: 1.351495
(Iteration 4091 / 4900) loss: 1.340985
(Iteration 4101 / 4900) loss: 0.839027
(Iteration 4111 / 4900) loss: 1.149286
(Iteration 4121 / 4900) loss: 1.158504
(Iteration 4131 / 4900) loss: 1.269635
(Iteration 4141 / 4900) loss: 1.140786
(Iteration 4151 / 4900) loss: 1.134655
(Iteration 4161 / 4900) loss: 1.157112
(Iteration 4171 / 4900) loss: 0.981483
(Iteration 4181 / 4900) loss: 1.194811
(Iteration 4191 / 4900) loss: 1.019904
(Iteration 4201 / 4900) loss: 1.377049
(Iteration 4211 / 4900) loss: 1.344512
(Iteration 4221 / 4900) loss: 1.160414
(Iteration 4231 / 4900) loss: 1.373117
(Iteration 4241 / 4900) loss: 1.142355
(Iteration 4251 / 4900) loss: 1.334775
(Iteration 4261 / 4900) loss: 1.234167
(Iteration 4271 / 4900) loss: 1.326630
(Iteration 4281 / 4900) loss: 1.322571
(Iteration 4291 / 4900) loss: 1.126674
(Iteration 4301 / 4900) loss: 1.511692
(Iteration 4311 / 4900) loss: 1.184628
(Iteration 4321 / 4900) loss: 1.346656
(Iteration 4331 / 4900) loss: 1.155393
(Iteration 4341 / 4900) loss: 1.082434
(Iteration 4351 / 4900) loss: 1.152772
(Iteration 4361 / 4900) loss: 1.307197
(Iteration 4371 / 4900) loss: 1.200055
(Iteration 4381 / 4900) loss: 1.320212
(Iteration 4391 / 4900) loss: 1.138229
(Iteration 4401 / 4900) loss: 1.130623
(Epoch 9 / 10) train acc: 0.567000; val_acc: 0.479000
(Iteration 4411 / 4900) loss: 1.295418
(Iteration 4421 / 4900) loss: 1.340685
(Iteration 4431 / 4900) loss: 1.228329
(Iteration 4441 / 4900) loss: 1.239255
(Iteration 4451 / 4900) loss: 1.212845
(Iteration 4461 / 4900) loss: 1.258915
(Iteration 4471 / 4900) loss: 1.311742
(Iteration 4481 / 4900) loss: 1.296166
(Iteration 4491 / 4900) loss: 1.269791
(Iteration 4501 / 4900) loss: 1.167893
(Iteration 4511 / 4900) loss: 1.110126
(Iteration 4521 / 4900) loss: 1.157010
(Iteration 4531 / 4900) loss: 1.402114
(Iteration 4541 / 4900) loss: 1.228235
(Iteration 4551 / 4900) loss: 1.116391
(Iteration 4561 / 4900) loss: 1.228602
(Iteration 4571 / 4900) loss: 1.081930
(Iteration 4581 / 4900) loss: 1.257248
(Iteration 4591 / 4900) loss: 1.131268
(Iteration 4601 / 4900) loss: 1.198317
(Iteration 4611 / 4900) loss: 1.480501
(Iteration 4621 / 4900) loss: 1.240263
(Iteration 4631 / 4900) loss: 1.280229
(Iteration 4641 / 4900) loss: 1.125298
(Iteration 4651 / 4900) loss: 1.122036
(Iteration 4661 / 4900) loss: 1.091655
(Iteration 4671 / 4900) loss: 1.470764
(Iteration 4681 / 4900) loss: 1.391047
```

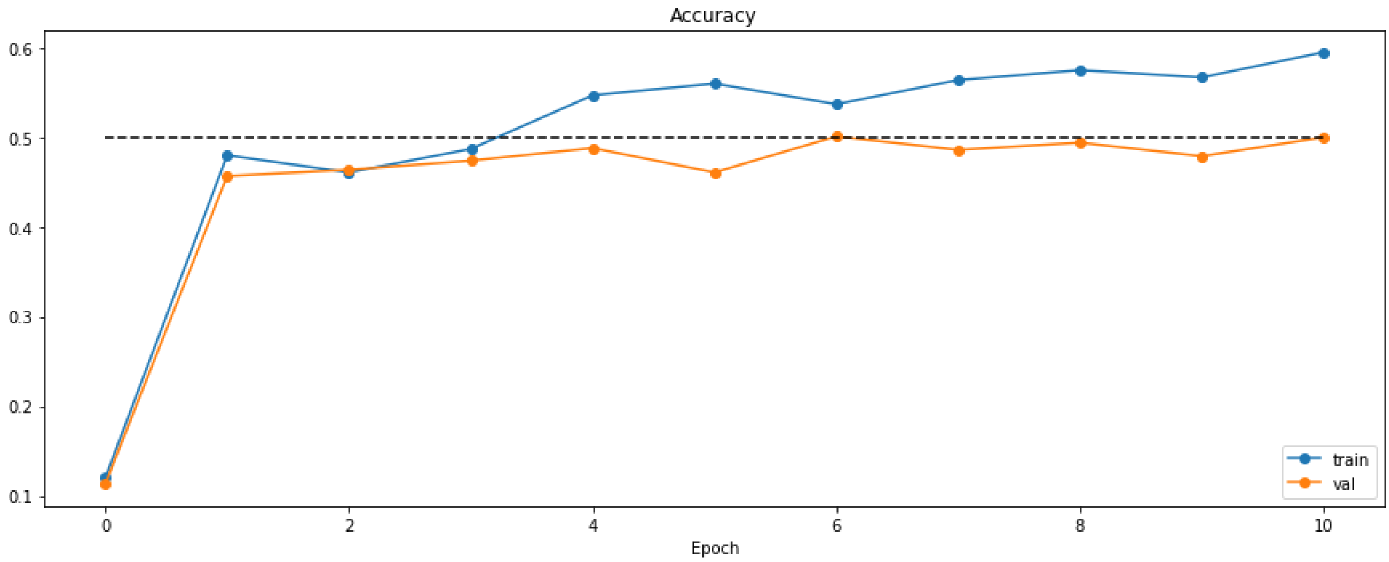
```
(Iteration 4691 / 4900) loss: 1.168193
(Iteration 4701 / 4900) loss: 1.090750
(Iteration 4711 / 4900) loss: 1.139016
(Iteration 4721 / 4900) loss: 1.021348
(Iteration 4731 / 4900) loss: 1.349709
(Iteration 4741 / 4900) loss: 1.395092
(Iteration 4751 / 4900) loss: 1.232225
(Iteration 4761 / 4900) loss: 1.336116
(Iteration 4771 / 4900) loss: 1.142804
(Iteration 4781 / 4900) loss: 1.158584
(Iteration 4791 / 4900) loss: 1.352555
(Iteration 4801 / 4900) loss: 1.269787
(Iteration 4811 / 4900) loss: 1.204076
(Iteration 4821 / 4900) loss: 1.323853
(Iteration 4831 / 4900) loss: 1.180223
(Iteration 4841 / 4900) loss: 1.246895
(Iteration 4851 / 4900) loss: 1.225930
(Iteration 4861 / 4900) loss: 1.433370
(Iteration 4871 / 4900) loss: 1.105787
(Iteration 4881 / 4900) loss: 1.192721
(Iteration 4891 / 4900) loss: 1.217191
(Epoch 10 / 10) train acc: 0.595000; val_acc: 0.500000
```

In [51]:

```
# Run this cell to visualize training loss and train / val accuracy

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

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



Multilayer network

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

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

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

Initial loss and gradient check

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

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

In [60]:

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

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

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

  # Most of the errors should be on the order of e-7 or smaller.
```

```
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  5.940411485412347
W1 relative error: 6.59e-09
W2 relative error: 5.91e-08
W3 relative error: 1.00e+00
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

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

In [64]:

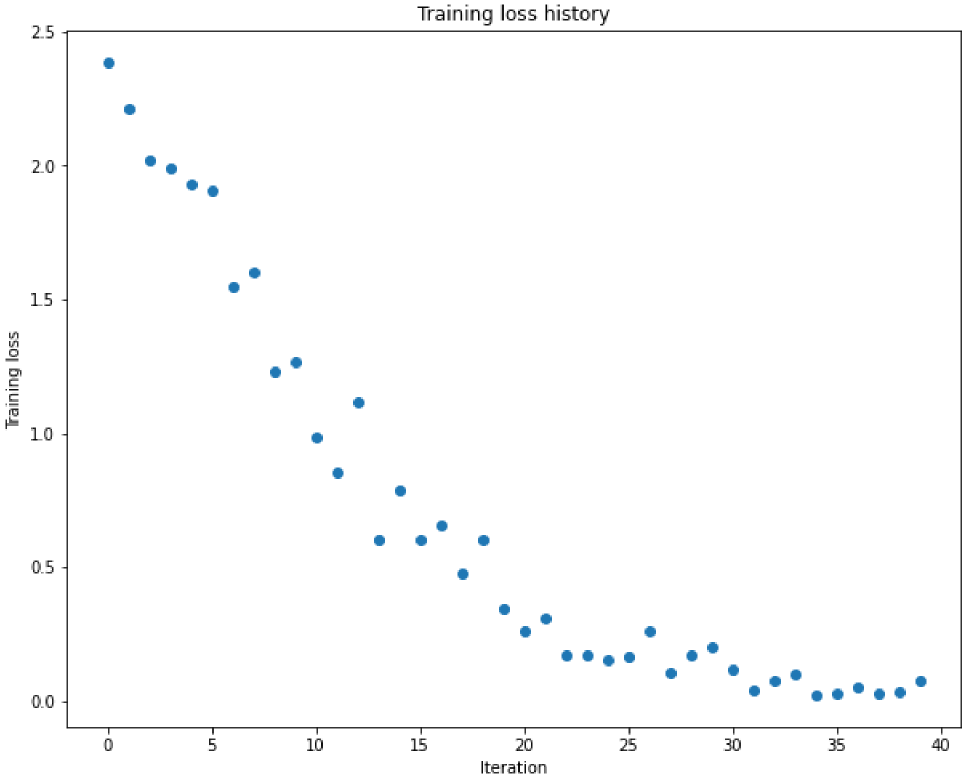
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-2   # Experiment with this!
learning_rate = 1e-2  # Experiment with this!
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.385904
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.112000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.113000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.109000
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.157000
(Epoch 4 / 20) train acc: 0.600000; val_acc: 0.123000
(Epoch 5 / 20) train acc: 0.700000; val_acc: 0.144000
(Iteration 11 / 40) loss: 0.986071
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.153000
(Epoch 7 / 20) train acc: 0.820000; val_acc: 0.168000
(Epoch 8 / 20) train acc: 0.900000; val_acc: 0.172000
(Epoch 9 / 20) train acc: 0.920000; val_acc: 0.187000
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.160000
(Iteration 21 / 40) loss: 0.258918
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.171000
(Epoch 12 / 20) train acc: 0.920000; val_acc: 0.182000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.189000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.194000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.177000
(Iteration 31 / 40) loss: 0.117595
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.174000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.188000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.189000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.181000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.182000
```



In [ ]: