Fully connected networks

In the previous notebook, you implemented a simple two-layer neural network class. However, this class is not modular. If you wanted to change the number of layers, you would need to write a new loss and gradient function. If you wanted to optimize the network with different optimizers, you'd need to write new training functions. If you wanted to incorporate regularizations, you'd have to modify the loss and gradient function.

Instead of having to modify functions each time, for the rest of the class, we'll work in a more modular framework where we define forward and backward layers that calculate losses and gradients respectively. Since the forward and backward layers share intermediate values that are useful for calculating both the loss and the gradient, we'll also have these function return "caches" which store useful intermediate values.

The goal is that through this modular design, we can build different sized neural networks for various applications.

In this HW #3, we'll define the basic architecture, and in HW #4, we'll build on this framework to implement different optimizers and regularizations (like BatchNorm and Dropout).

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).

Modular layers

This notebook will build modular layers in the following manner. First, there will be a forward pass for a given layer with inputs (`x`) and return the output of that layer (`out`) as well as cached variables (`cache`) that will be used to calculate the gradient in the backward pass.

```
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 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
```

In [30]:
```
## 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_numerical_grad
ient_array
from cs231n.solver import Solver

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

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%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 [31]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

Linear layers

In this section, we'll implement the forward and backward pass for the linear layers.

The linear layer forward pass is the function `affine_forward` in `nndl/layers.py` and the backward pass is `affine_backward`.

After you have implemented these, test your implementation by running the cell below.

Affine layer forward pass

Implement `affine_forward` and then test your code by running the following cell.

In [32]:
```
# 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), outp
ut_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 1e-9.
print('Testing affine_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

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

Affine layer backward pass

Implement `affine_backward` and then test your code by running the following cell.

In [33]:
```
# Test the affine_backward function

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 1e-10
print('Testing affine_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_backward function:
dx error: 2.0102720432483927e-10
dw error: 1.786902592840743e-10
db error: 5.168732824993997e-11
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

Implement the `relu_forward` function in `nndl/layers.py` and then test your code by running the following cell.

In [34]:
```
# 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 around 1e-8
print('Testing relu_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

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

ReLU backward pass

Implement the `relu_backward` function in `nndl/layers.py` and then test your code by running the following cell.

In [35]:
```
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 around 1e-12
print('Testing relu_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing relu_backward function:
dx error: 3.27562144014014e-12
```

Combining the affine and ReLU layers

Often times, an affine layer will be followed by a ReLU layer. So let's make one that puts them together. Layers that are combined are stored in `nndl/layer_utils.py`.

Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nndl/layer_utils.py`. Take a look at them to make sure you understand what's going on. Then run the following cell to ensure its implemented correctly.

In [36]:
```
from nndl.layer_utils import affine_relu_forward, affine_relu_backward

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)

print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error: 1.647876847124382e-10
dw error: 7.367935047019492e-10
db error: 6.506831444739067e-11
```

Softmax and SVM losses

You've already implemented these, so we have written these in `layers.py` . The following code will ensure they are working correctly.

In [37]:
```
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: svm_loss(x, y)[0], x, verbose=False
)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))

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

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing svm_loss:
loss: 9.000898228652648
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.302675382264634
dx error: 1.0603610683853635e-08
```

Implementation of a two-layer NN

In `nndl/fc_net.py` , implement the class `TwoLayerNet` which uses the layers you made here. When you have finished, the following cell will test your implementation.

In [38]:
```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dims=H, num_classes=C, weight_scale=st
d)

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, 15.33
206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49
994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66
781506,  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
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = {}'.format(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('{} relative error: {}'.format(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: 1.8336562786695002e-08
W2 relative error: 3.201560569143183e-10
b1 relative error: 9.828315204644842e-09
b2 relative error: 4.329134954569865e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279152310200606e-07
W2 relative error: 2.8508510893102143e-08
b1 relative error: 1.564679947504764e-08
b2 relative error: 9.089617896905665e-10
```

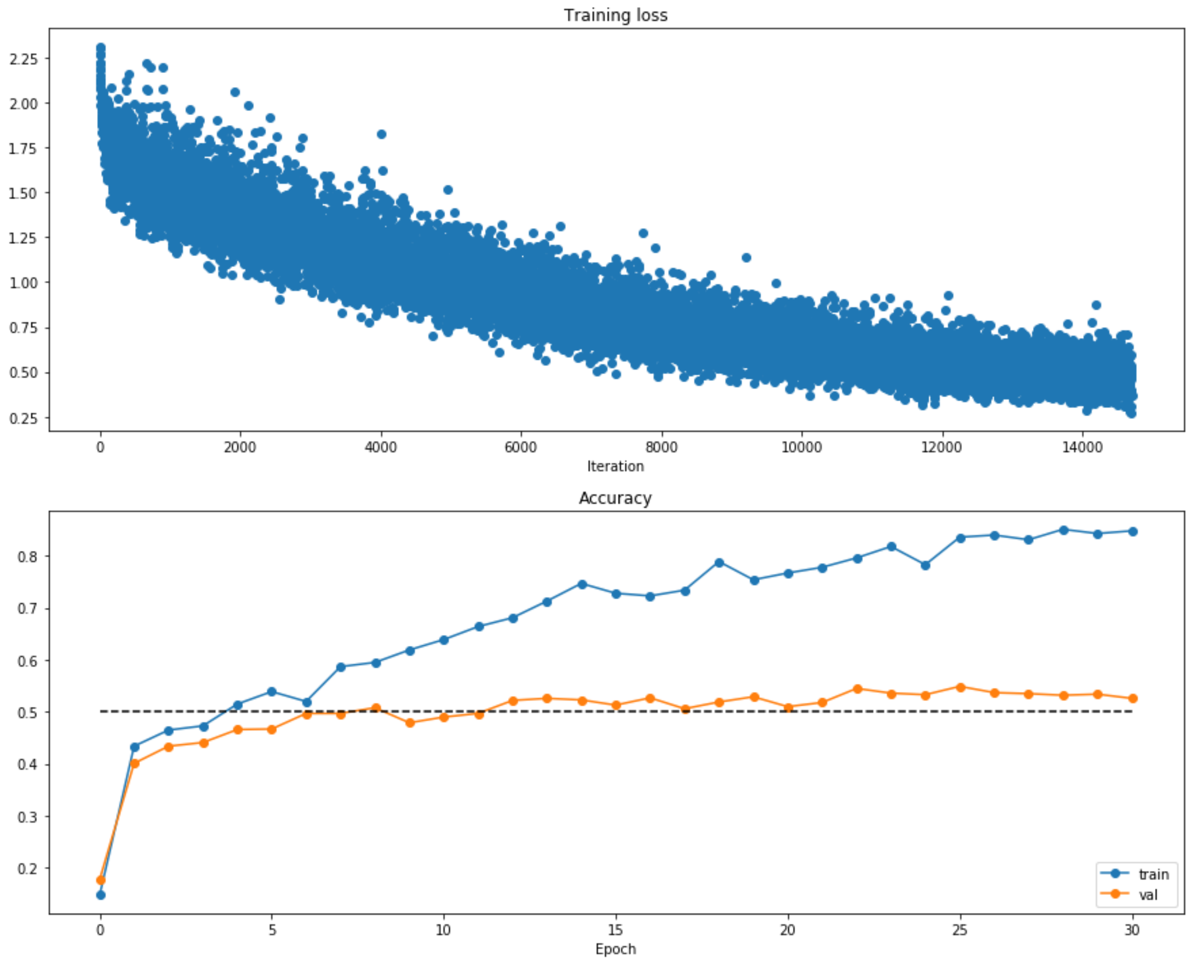
Solver

We will now use the cs231n Solver class to train these networks. Familiarize yourself with the API in `cs231n/solver.py` . After you have done so, declare an instance of a TwoLayerNet with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least 50%.

In [39]:
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 50%.  We won't have you optimize this further
#   since you did it in the previous notebook.
#
# ================================================================ #

model = TwoLayerNet(input_dim = 3*32*32, hidden_dims = 200, num_classes = 10,
weight_scale = 1e-3)
solver = Solver(model, data, update_rule = 'sgd', optim_config = {'learning_ra
te': 0.0018889},
               lr_decay = 0.9125, num_epochs = 30, batch_size = 100, print_eve
ry = 100000)
solver.train()

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

```
(Iteration 1 / 14700) loss: 2.309769
(Epoch 0 / 30) train acc: 0.148000; val_acc: 0.178000
(Epoch 1 / 30) train acc: 0.434000; val_acc: 0.401000
(Epoch 2 / 30) train acc: 0.465000; val_acc: 0.434000
(Epoch 3 / 30) train acc: 0.473000; val_acc: 0.441000
(Epoch 4 / 30) train acc: 0.515000; val_acc: 0.466000
(Epoch 5 / 30) train acc: 0.539000; val_acc: 0.467000
(Epoch 6 / 30) train acc: 0.520000; val_acc: 0.497000
(Epoch 7 / 30) train acc: 0.587000; val_acc: 0.497000
(Epoch 8 / 30) train acc: 0.595000; val_acc: 0.508000
(Epoch 9 / 30) train acc: 0.619000; val_acc: 0.479000
(Epoch 10 / 30) train acc: 0.639000; val_acc: 0.490000
(Epoch 11 / 30) train acc: 0.664000; val_acc: 0.497000
(Epoch 12 / 30) train acc: 0.681000; val_acc: 0.522000
(Epoch 13 / 30) train acc: 0.713000; val_acc: 0.526000
(Epoch 14 / 30) train acc: 0.747000; val_acc: 0.523000
(Epoch 15 / 30) train acc: 0.728000; val_acc: 0.513000
(Epoch 16 / 30) train acc: 0.723000; val_acc: 0.527000
(Epoch 17 / 30) train acc: 0.734000; val_acc: 0.506000
(Epoch 18 / 30) train acc: 0.789000; val_acc: 0.519000
(Epoch 19 / 30) train acc: 0.754000; val_acc: 0.529000
(Epoch 20 / 30) train acc: 0.767000; val_acc: 0.510000
(Epoch 21 / 30) train acc: 0.778000; val_acc: 0.518000
(Epoch 22 / 30) train acc: 0.796000; val_acc: 0.545000
(Epoch 23 / 30) train acc: 0.818000; val_acc: 0.536000
(Epoch 24 / 30) train acc: 0.783000; val_acc: 0.533000
(Epoch 25 / 30) train acc: 0.836000; val_acc: 0.549000
(Epoch 26 / 30) train acc: 0.840000; val_acc: 0.537000
(Epoch 27 / 30) train acc: 0.831000; val_acc: 0.535000
(Epoch 28 / 30) train acc: 0.851000; val_acc: 0.532000
(Epoch 29 / 30) train acc: 0.843000; val_acc: 0.534000
(Epoch 30 / 30) train acc: 0.848000; val_acc: 0.526000
```

In [40]:
```
# 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 Neural Network

Now, we implement a multi-layer neural network.

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

Implement the initialization, the forward pass, and the backward pass. There will be lines for batchnorm and dropout layers and caches; ignore these all for now. That'll be in assignment #4.

In [41]:
```
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 = {}'.format(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: {}'.format(loss))

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

```
Running check with reg = 0
Initial loss: 3.3663538412813647
W1 relative error: 5.2888338248209637e-08
W2 relative error: 1.6791539477084933e-07
b1 relative error: 1.0889278181160834e-09
b2 relative error: 6.782527079089013e-10
Running check with reg = 3.14
Initial loss: 4.656960094658376
W1 relative error: 7.551965269664439e-08
W2 relative error: 4.9816354847798 69e-08
b1 relative error: 2.541723727607985e-08
b2 relative error: 1.2943627684305898e-09
```

In [50]:
```
# Use the three layer neural network to overfit a small dataset.

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'],
}


#### !!!!!!
# Play around with the weight_scale and learning_rate so that you can overfit
 a small dataset.
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 0.00004
learning_rate = 0.0010

model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=200, 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()
```

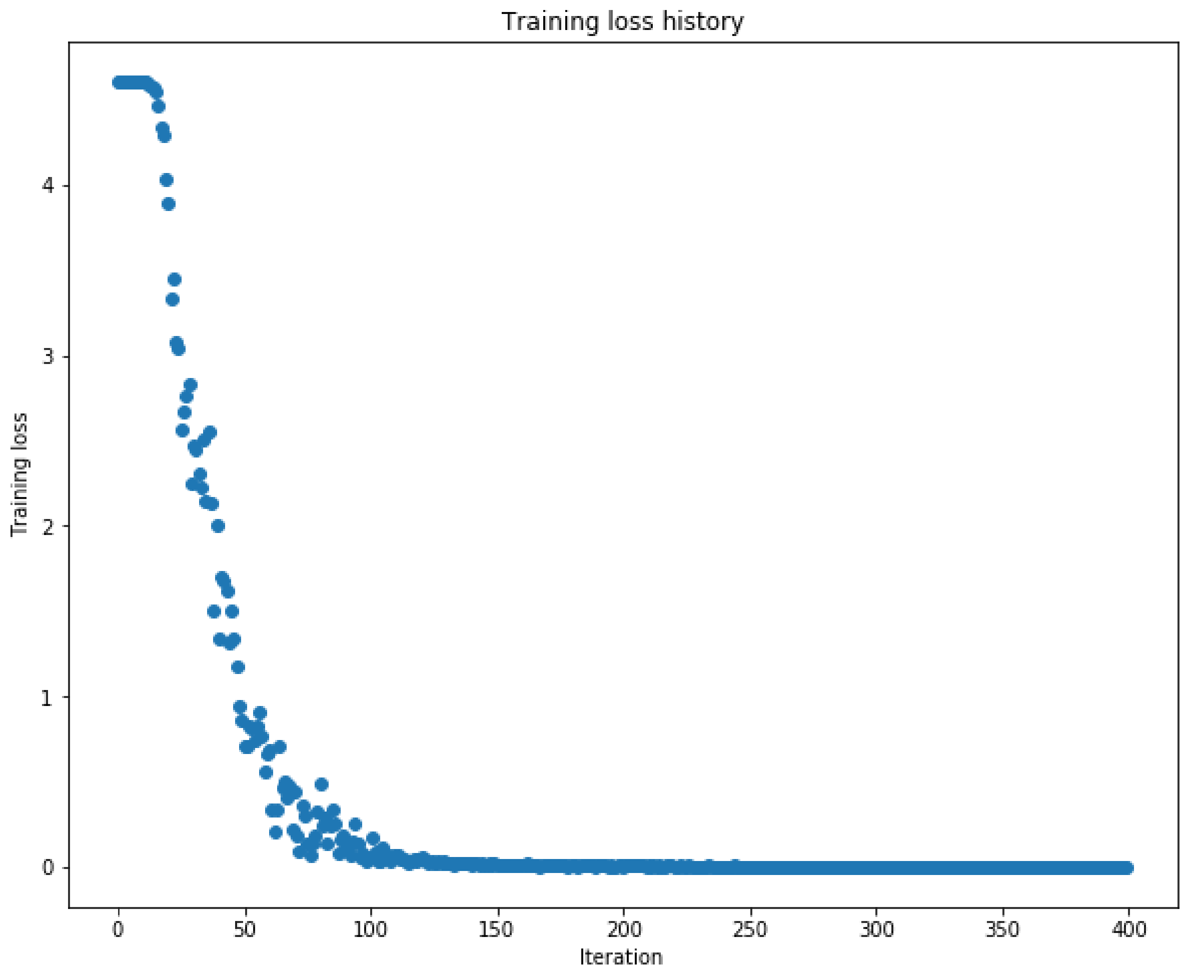
```
(Iteration 1 / 400) loss: 4.605172
(Epoch 0 / 200) train acc: 0.220000; val_acc: 0.107000
(Epoch 1 / 200) train acc: 0.220000; val_acc: 0.092000
(Epoch 2 / 200) train acc: 0.280000; val_acc: 0.154000
(Epoch 3 / 200) train acc: 0.340000; val_acc: 0.140000
(Epoch 4 / 200) train acc: 0.220000; val_acc: 0.105000
(Epoch 5 / 200) train acc: 0.220000; val_acc: 0.100000
(Iteration 11 / 400) loss: 4.599741
(Epoch 6 / 200) train acc: 0.260000; val_acc: 0.122000
(Epoch 7 / 200) train acc: 0.200000; val_acc: 0.135000
(Epoch 8 / 200) train acc: 0.220000; val_acc: 0.141000
(Epoch 9 / 200) train acc: 0.220000; val_acc: 0.085000
(Epoch 10 / 200) train acc: 0.200000; val_acc: 0.116000
(Iteration 21 / 400) loss: 3.889044
(Epoch 11 / 200) train acc: 0.240000; val_acc: 0.088000
(Epoch 12 / 200) train acc: 0.280000; val_acc: 0.149000
(Epoch 13 / 200) train acc: 0.260000; val_acc: 0.128000
(Epoch 14 / 200) train acc: 0.220000; val_acc: 0.111000
(Epoch 15 / 200) train acc: 0.280000; val_acc: 0.154000
(Iteration 31 / 400) loss: 2.468571
(Epoch 16 / 200) train acc: 0.320000; val_acc: 0.141000
(Epoch 17 / 200) train acc: 0.340000; val_acc: 0.159000
(Epoch 18 / 200) train acc: 0.320000; val_acc: 0.158000
(Epoch 19 / 200) train acc: 0.380000; val_acc: 0.169000
(Epoch 20 / 200) train acc: 0.500000; val_acc: 0.160000
(Iteration 41 / 400) loss: 1.344916
(Epoch 21 / 200) train acc: 0.560000; val_acc: 0.166000
(Epoch 22 / 200) train acc: 0.660000; val_acc: 0.169000
(Epoch 23 / 200) train acc: 0.700000; val_acc: 0.184000
(Epoch 24 / 200) train acc: 0.700000; val_acc: 0.173000
(Epoch 25 / 200) train acc: 0.780000; val_acc: 0.181000
(Iteration 51 / 400) loss: 0.706996
(Epoch 26 / 200) train acc: 0.800000; val_acc: 0.194000
(Epoch 27 / 200) train acc: 0.840000; val_acc: 0.201000
(Epoch 28 / 200) train acc: 0.840000; val_acc: 0.175000
(Epoch 29 / 200) train acc: 0.840000; val_acc: 0.192000
(Epoch 30 / 200) train acc: 0.820000; val_acc: 0.171000
(Iteration 61 / 400) loss: 0.688226
(Epoch 31 / 200) train acc: 0.900000; val_acc: 0.176000
(Epoch 32 / 200) train acc: 0.900000; val_acc: 0.183000
(Epoch 33 / 200) train acc: 0.840000; val_acc: 0.162000
(Epoch 34 / 200) train acc: 0.900000; val_acc: 0.188000
(Epoch 35 / 200) train acc: 0.860000; val_acc: 0.163000
(Iteration 71 / 400) loss: 0.445107
(Epoch 36 / 200) train acc: 0.920000; val_acc: 0.178000
(Epoch 37 / 200) train acc: 0.960000; val_acc: 0.170000
(Epoch 38 / 200) train acc: 0.960000; val_acc: 0.179000
(Epoch 39 / 200) train acc: 0.940000; val_acc: 0.186000
(Epoch 40 / 200) train acc: 0.940000; val_acc: 0.160000
(Iteration 81 / 400) loss: 0.494688
(Epoch 41 / 200) train acc: 0.960000; val_acc: 0.170000
(Epoch 42 / 200) train acc: 0.940000; val_acc: 0.186000
(Epoch 43 / 200) train acc: 0.960000; val_acc: 0.170000
(Epoch 44 / 200) train acc: 0.940000; val_acc: 0.191000
(Epoch 45 / 200) train acc: 0.980000; val_acc: 0.180000
(Iteration 91 / 400) loss: 0.109107
(Epoch 46 / 200) train acc: 0.960000; val_acc: 0.170000
```

```
(Epoch 47 / 200) train acc: 0.960000; val_acc: 0.181000
(Epoch 48 / 200) train acc: 1.000000; val_acc: 0.192000
(Epoch 49 / 200) train acc: 0.980000; val_acc: 0.189000
(Epoch 50 / 200) train acc: 1.000000; val_acc: 0.175000
(Iteration 101 / 400) loss: 0.059865
(Epoch 51 / 200) train acc: 0.980000; val_acc: 0.194000
(Epoch 52 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 53 / 200) train acc: 1.000000; val_acc: 0.184000
(Epoch 54 / 200) train acc: 1.000000; val_acc: 0.188000
(Epoch 55 / 200) train acc: 1.000000; val_acc: 0.189000
(Iteration 111 / 400) loss: 0.054751
(Epoch 56 / 200) train acc: 1.000000; val_acc: 0.185000
(Epoch 57 / 200) train acc: 1.000000; val_acc: 0.186000
(Epoch 58 / 200) train acc: 1.000000; val_acc: 0.183000
(Epoch 59 / 200) train acc: 1.000000; val_acc: 0.186000
(Epoch 60 / 200) train acc: 1.000000; val_acc: 0.188000
(Iteration 121 / 400) loss: 0.063551
(Epoch 61 / 200) train acc: 1.000000; val_acc: 0.187000
(Epoch 62 / 200) train acc: 1.000000; val_acc: 0.184000
(Epoch 63 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 64 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 65 / 200) train acc: 1.000000; val_acc: 0.184000
(Iteration 131 / 400) loss: 0.024325
(Epoch 66 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 67 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 68 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 69 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 70 / 200) train acc: 1.000000; val_acc: 0.182000
(Iteration 141 / 400) loss: 0.016950
(Epoch 71 / 200) train acc: 1.000000; val_acc: 0.184000
(Epoch 72 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 73 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 74 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 75 / 200) train acc: 1.000000; val_acc: 0.174000
(Iteration 151 / 400) loss: 0.015496
(Epoch 76 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 77 / 200) train acc: 1.000000; val_acc: 0.171000
(Epoch 78 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 79 / 200) train acc: 1.000000; val_acc: 0.171000
(Epoch 80 / 200) train acc: 1.000000; val_acc: 0.172000
(Iteration 161 / 400) loss: 0.012433
(Epoch 81 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 82 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 83 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 84 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 85 / 200) train acc: 1.000000; val_acc: 0.170000
(Iteration 171 / 400) loss: 0.015077
(Epoch 86 / 200) train acc: 1.000000; val_acc: 0.170000
(Epoch 87 / 200) train acc: 1.000000; val_acc: 0.173000
(Epoch 88 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 89 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 90 / 200) train acc: 1.000000; val_acc: 0.172000
(Iteration 181 / 400) loss: 0.014952
(Epoch 91 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 92 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 93 / 200) train acc: 1.000000; val_acc: 0.175000
(Epoch 94 / 200) train acc: 1.000000; val_acc: 0.175000
```

```
(Epoch 95 / 200) train acc: 1.000000; val_acc: 0.176000
(Iteration 191 / 400) loss: 0.009590
(Epoch 96 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 97 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 98 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 99 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 100 / 200) train acc: 1.000000; val_acc: 0.180000
(Iteration 201 / 400) loss: 0.006620
(Epoch 101 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 102 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 103 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 104 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 105 / 200) train acc: 1.000000; val_acc: 0.183000
(Iteration 211 / 400) loss: 0.009187
(Epoch 106 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 107 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 108 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 109 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 110 / 200) train acc: 1.000000; val_acc: 0.182000
(Iteration 221 / 400) loss: 0.005830
(Epoch 111 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 112 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 113 / 200) train acc: 1.000000; val_acc: 0.173000
(Epoch 114 / 200) train acc: 1.000000; val_acc: 0.176000
(Epoch 115 / 200) train acc: 1.000000; val_acc: 0.174000
(Iteration 231 / 400) loss: 0.007205
(Epoch 116 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 117 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 118 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 119 / 200) train acc: 1.000000; val_acc: 0.176000
(Epoch 120 / 200) train acc: 1.000000; val_acc: 0.174000
(Iteration 241 / 400) loss: 0.005731
(Epoch 121 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 122 / 200) train acc: 1.000000; val_acc: 0.174000
(Epoch 123 / 200) train acc: 1.000000; val_acc: 0.175000
(Epoch 124 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 125 / 200) train acc: 1.000000; val_acc: 0.178000
(Iteration 251 / 400) loss: 0.005841
(Epoch 126 / 200) train acc: 1.000000; val_acc: 0.173000
(Epoch 127 / 200) train acc: 1.000000; val_acc: 0.171000
(Epoch 128 / 200) train acc: 1.000000; val_acc: 0.173000
(Epoch 129 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 130 / 200) train acc: 1.000000; val_acc: 0.171000
(Iteration 261 / 400) loss: 0.006214
(Epoch 131 / 200) train acc: 1.000000; val_acc: 0.176000
(Epoch 132 / 200) train acc: 1.000000; val_acc: 0.172000
(Epoch 133 / 200) train acc: 1.000000; val_acc: 0.175000
(Epoch 134 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 135 / 200) train acc: 1.000000; val_acc: 0.175000
(Iteration 271 / 400) loss: 0.004203
(Epoch 136 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 137 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 138 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 139 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 140 / 200) train acc: 1.000000; val_acc: 0.179000
(Iteration 281 / 400) loss: 0.004103
(Epoch 141 / 200) train acc: 1.000000; val_acc: 0.179000
```

```
(Epoch 142 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 143 / 200) train acc: 1.000000; val_acc: 0.175000
(Epoch 144 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 145 / 200) train acc: 1.000000; val_acc: 0.181000
(Iteration 291 / 400) loss: 0.003961
(Epoch 146 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 147 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 148 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 149 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 150 / 200) train acc: 1.000000; val_acc: 0.176000
(Iteration 301 / 400) loss: 0.005293
(Epoch 151 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 152 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 153 / 200) train acc: 1.000000; val_acc: 0.182000
(Epoch 154 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 155 / 200) train acc: 1.000000; val_acc: 0.179000
(Iteration 311 / 400) loss: 0.003632
(Epoch 156 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 157 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 158 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 159 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 160 / 200) train acc: 1.000000; val_acc: 0.177000
(Iteration 321 / 400) loss: 0.003431
(Epoch 161 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 162 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 163 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 164 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 165 / 200) train acc: 1.000000; val_acc: 0.180000
(Iteration 331 / 400) loss: 0.003142
(Epoch 166 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 167 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 168 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 169 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 170 / 200) train acc: 1.000000; val_acc: 0.179000
(Iteration 341 / 400) loss: 0.004564
(Epoch 171 / 200) train acc: 1.000000; val_acc: 0.176000
(Epoch 172 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 173 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 174 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 175 / 200) train acc: 1.000000; val_acc: 0.178000
(Iteration 351 / 400) loss: 0.003547
(Epoch 176 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 177 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 178 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 179 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 180 / 200) train acc: 1.000000; val_acc: 0.178000
(Iteration 361 / 400) loss: 0.003727
(Epoch 181 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 182 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 183 / 200) train acc: 1.000000; val_acc: 0.178000
(Epoch 184 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 185 / 200) train acc: 1.000000; val_acc: 0.177000
(Iteration 371 / 400) loss: 0.002589
(Epoch 186 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 187 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 188 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 189 / 200) train acc: 1.000000; val_acc: 0.179000
```

```
(Epoch 190 / 200) train acc: 1.000000; val_acc: 0.180000
(Iteration 381 / 400) loss: 0.003914
(Epoch 191 / 200) train acc: 1.000000; val_acc: 0.180000
(Epoch 192 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 193 / 200) train acc: 1.000000; val_acc: 0.181000
(Epoch 194 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 195 / 200) train acc: 1.000000; val_acc: 0.178000
(Iteration 391 / 400) loss: 0.003073
(Epoch 196 / 200) train acc: 1.000000; val_acc: 0.177000
(Epoch 197 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 198 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 199 / 200) train acc: 1.000000; val_acc: 0.179000
(Epoch 200 / 200) train acc: 1.000000; val_acc: 0.179000
```



```
In [ ]:
```

```
In [ ]:
```