① $\mathcal{L} = \frac{1}{2}\|W^TWx - x\|^2$ $\quad x \in \mathbb{R}^n \quad m < n$ (fewer rows than cols)

$W \in \mathbb{R}^{m \times n}$

a) $W^T \in \mathbb{R}^{n \times m}$

$W^TW \in \mathbb{R}^{n \times n}$

$Wx \in \mathbb{R}^{m \times 1}$

$$m\downarrow\left[\quad \xrightarrow{n} \quad\right]\left[\begin{matrix} | \\ n \\ | \end{matrix}\right]$$

$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix}\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

$$\begin{bmatrix} w_{11}^2 + w_{21}^2 & w_{11}w_{12} + w_{21}w_{22} \\ w_{11}w_{12} - w_{21}w_{22} & w_{12}^2 + w_{22}^2 \end{bmatrix}$$

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{n}\left|\sum_{j=1}^{m} w_{ij}w_{ji}x_i - x_i\right|^2$$

W^TW creates a square matrix so that some predicted value W^TWx and ground truth x have the same dimension. This minimization is essentially computing a covariance that can be decomposed into an eigenvector matrix V and eigenvalue matrix λ:

$$L = \sum_i (W^TWx_i - x_i)^T(W^TWx_i - x_i)$$

$$= \sum_i tr\left[(W^TW - I)x_ix_i^T(W^TW - I)\right]$$

$$= tr\left[(W^TW - I)\sum_i(x_ix_i^T)(W^TW - I)\right]$$

$$= tr\left[(W^TW - I)\underbrace{V\lambda V}(W^TW - I)\right]$$

this is our eigenvector/eigenvalue product representing covariance.

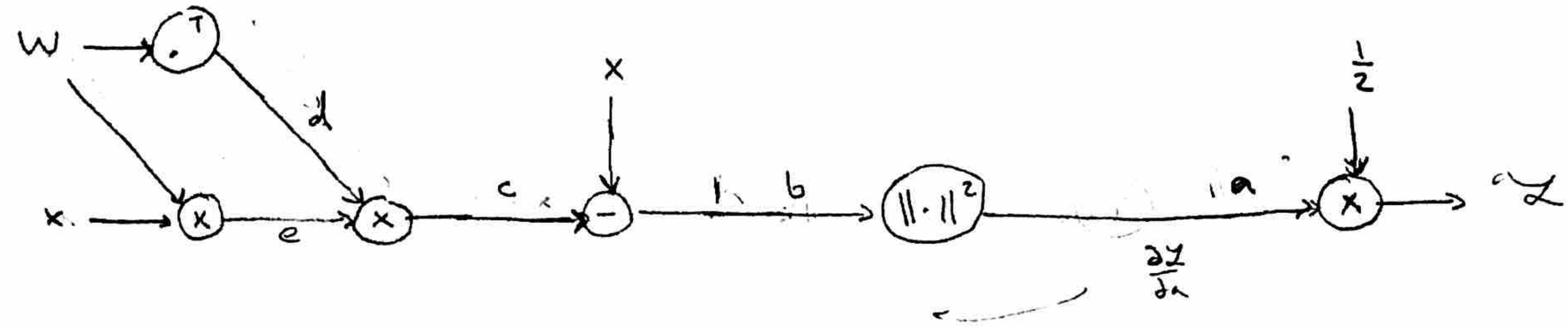
In essence, we are minimizing weights such that we only keep directions of strongest variance (largest eigenvalues). This is similar to PCA. By preserving the terms corresponding to largest variance, we capture the most essential/influential information.

b)

$c = W^T W x$
$(n\times n)(n\times 1)$

$\mathcal{L} \in \mathbb{R}^1$

$d \in \mathbb{R}^{n\times m}$
$e \in \mathbb{R}^{m\times 1}$
$c \in \mathbb{R}^{n\times 1}$
$b \in \mathbb{R}^{n\times 1}$
$a \in \mathbb{R}^1$



c) One path is through the transpose operation and the other is directly to W. We need to consider:

$$\frac{\partial \mathcal{L}}{\partial W} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial W^T}$$

The final solution will involve both of these paths summed:

$$\frac{\partial \mathcal{L}}{\partial W} = \left(\frac{\partial \mathcal{L}}{\partial d}\right)^T + \frac{\partial \mathcal{L}}{\partial e}$$

$$\left(\frac{\partial \mathcal{L}}{\partial d}\right)^T = \frac{\partial \mathcal{L}}{\partial d^T}\Big|_{d=W}$$

d) We want $\frac{\partial f}{\partial W}$

$$\mathcal{L} = \frac{1}{2} a$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial a} = \frac{1}{2}}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial b}$$

$$a = \|b\|^2$$

$$\frac{\partial a}{\partial b} = \frac{\partial}{\partial b}\|b\|^2 = \frac{\partial}{\partial b_j}\sum_{k=1}^{n} b_k^2$$

$$= \sum_{k=1}^{n} \frac{\partial}{\partial b_j} b_k^2 = \begin{cases} 0 & \text{if } j \neq k \\ 2b_k & \text{if } j = k \end{cases}$$

$$= 2b_j$$

$$\frac{\partial a}{\partial b} = 2b$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{\cancel{2}}(\cancel{2}b)$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = b}$$

$$\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial c}$$

$$b = c - x$$

$$\frac{\partial b}{\partial c} = \frac{\partial}{\partial c}(c - x)$$

$$\frac{\partial \mathcal{L}}{\partial c} = \frac{\partial \mathcal{L}}{\partial b}\left(\cancel{\frac{\partial}{\partial c} c}\right)^{1}$$

$$\frac{\partial \mathcal{L}}{\partial d} = \frac{\partial \mathcal{L}}{\partial c}\frac{\partial c}{\partial d}, \quad d = w^T$$

$$= \left(\frac{\partial}{\partial c} c\right)\frac{\partial c}{\partial d}$$

$$c = de \Rightarrow c = w^T W x, \quad d = w^T$$

$$\frac{\partial c}{\partial d} = \frac{\partial (de)}{\partial d} = \frac{\partial (w^T W x)}{\partial w^T}$$

$$w^T W x \in \mathbb{R}^m, \quad \frac{\partial (w^T W x)}{\partial w^T} \in \mathbb{R}^{n \times m \times n}$$

$$\frac{\partial c}{\partial d} = (Wx)^T$$

$$\frac{\partial \mathcal{L}}{\partial e}_{(m\times 1)} = \frac{\partial c}{\partial e}_{(m\times n)}\frac{\partial \mathcal{L}}{\partial c}_{(n\times 1)}$$

$$c = w^T W x, \quad e = Wx$$

$$c \in \mathbb{R}^n, \quad e \in \mathbb{R}^{m \times 1}$$

$$\frac{\partial c}{\partial e} = \frac{\partial (w^T W x)}{\partial Wx}$$

$$\frac{\partial \mathcal{L}}{\partial e} = \frac{\partial c}{\partial e}\frac{\partial \mathcal{L}}{\partial c}$$ — scalar

$$\frac{\partial \mathcal{L}}{\partial e} = Wx^T \frac{\partial c}{\partial e}$$

$$\frac{\partial \mathcal{L}}{\partial w^T} = \left(\frac{\partial \mathcal{L}}{\partial w}\right)^T$$

$$\frac{\partial \mathcal{L}}{\partial W}_{(m\times n)} = \left(\frac{\partial \mathcal{L}}{\partial d}\right)^T + \frac{\partial \mathcal{L}}{\partial e}$$

$$= \left(\frac{\partial \mathcal{L}}{\partial c}\frac{\partial c}{\partial d}\right)^T + \frac{\partial c}{\partial e}\frac{\partial \mathcal{L}}{\partial c}$$

$$= \left(\frac{\partial \mathcal{L}}{\partial b}(Wx)^T\right)^T + Wx^T\frac{\partial \mathcal{L}}{\partial c}$$

$$= \underbrace{(w^T W x - x)^T}_{1\times n}(Wx) + \underbrace{W}_{(m\times n)}\underbrace{(w^T W x - x)}_{(1\times n)^T}\underbrace{x^T}_{n\times 1}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial W} = (Wx)(w^T W x - x)^T + W(w^T W x - x)x^T}$$

This is the 2-layer neural network workbook for ECE 239AS Assignment #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a two layer neural network.

In [133]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%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
```

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass

In [134]:
```
from nndl.neural_net import TwoLayerNet
```

In [135]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

Compute forward pass scores

In [136]:

```
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.38123121099e-08
```

Forward pass loss

In [137]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

print(loss)

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
1.07169612386
Difference between your loss and correct loss:
0.0
```

In [138]:
```
print(loss)
```

```
1.07169612386
```

Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

In [139]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the ba
ckward pass.
# If your implementation is correct, the difference between the numeri
c and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1,
and b2.

loss, grads = net.loss(X, y, reg=0.05)
#print(grads.shape)
# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name]
, verbose=False)
    print(grads[param_name].shape, param_grad_num.shape)
    print('{} max relative error: {}'.format(param_name, rel_error(par
am_grad_num, grads[param_name])))
```

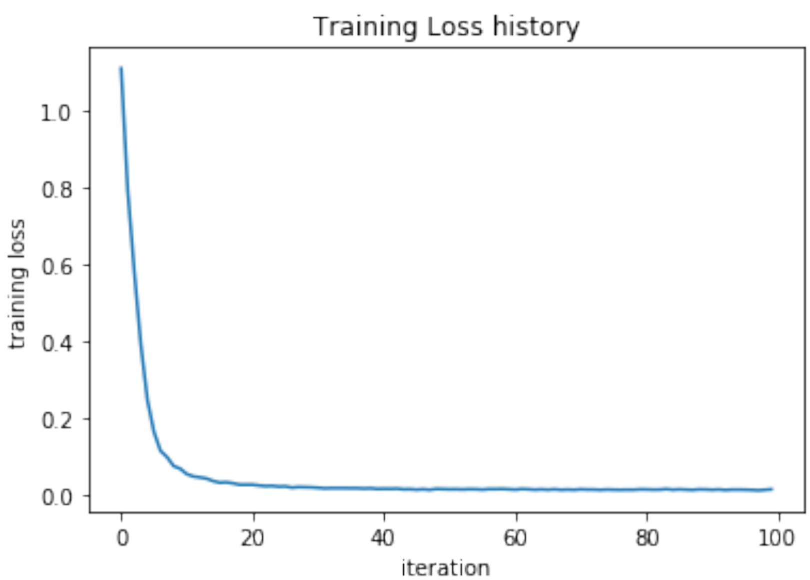
```
(3, 10) (3, 10)
W2 max relative error: 2.9632227682005116e-10
(3,) (3,)
b2 max relative error: 1.2482624742512528e-09
(10, 4) (10, 4)
W1 max relative error: 1.283285235125835e-09
(10,) (10,)
b1 max relative error: 3.172680092703762e-09
```

Training the network

Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

In [140]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.0144978645878
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [141]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test
=1000, verbose=True):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to p
repare
    it for the two-layer neural net classifier. These are the same ste
ps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

In [142]:
```
input_size = 32 * 32 * 3


def train_net(batch_size=200, learning_rate=1e-4, num_iters=1000, reg=
0.25):
    hidden_size = 50
    num_classes = 10
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    # Train the network
    stats = net.train(X_train, y_train, X_val, y_val,
                num_iters=num_iters, batch_size=batch_size,
                learning_rate=learning_rate, learning_rate_decay=0.95,
                reg=reg, verbose=False)

    # Predict on the validation set
    val_acc = (net.predict(X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)

    # Save this net as the variable subopt_net for later comparison.
    subopt_net = net
    return val_acc, stats, subopt_net
```

Questions:

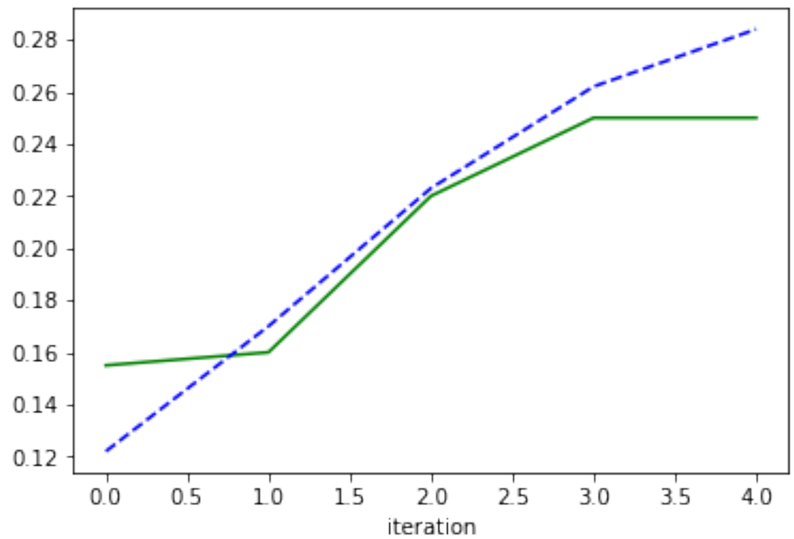
The training accuracy isn't great.

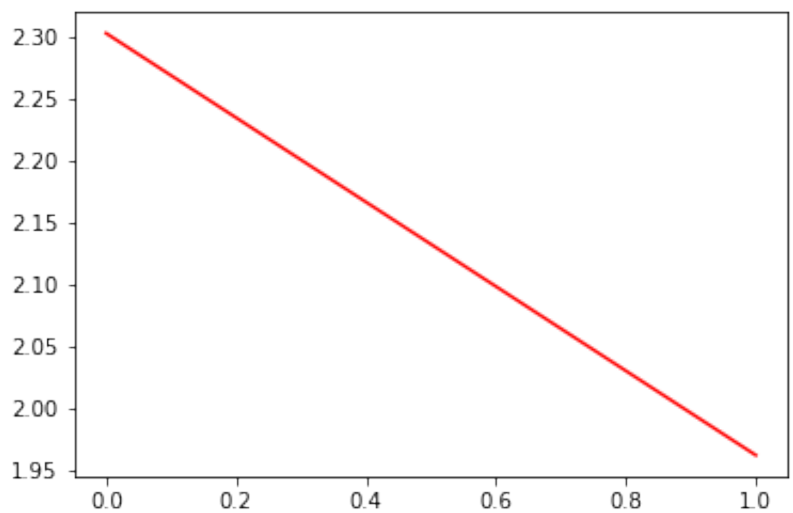
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

In [144]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies
# plot the loss history

#plt.plot(stats['loss_history'], 'r')
val_acc, stats, net = train_net()
subsampled_loss_history = stats['loss_history'][0::batch_size]
#print(subsampled_loss_history)


plt.plot(stats['train_acc_history'], 'g-')
plt.plot(stats['val_acc_history'], 'b--')
plt.xlabel('iteration')
#plt.ylabel('training loss')
#plt.title('Training Loss history')
plt.show()

plt.plot(subsampled_loss_history, 'r')

plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Validation accuracy:  0.28
```





Answers:

(1) It appears as if both training accuracy and validation accuracy are increasing for 1000 iterations. Since we haven't reached the point at which validation accuracy decreases and training accuracy increases (indicating overfitting), it's likely that SGD has not performed enough iterations to find a local min. It's also worth noting that the training and validation error seem to match. Typically there is a spread as the model becomes better at memorizing the data with extended training. If the model is overly complex, this spread will be large. If the model is not complex enough, a spread might just not exist.

In addition, the loss is decreasing linearly rather than exponentially decaying. It's possible that the learning rate might not be high enough.

(2) The first thing to optimize is the number of iterations. If we never reach the min, the accuracy of the model will be low. After this, learning rate can be adjusted as a hyperparameter if it is observed that the gradient is not stabilizing.

Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

In [146]:
```
best_net = None # store the best model into this
import itertools
# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied
by:
#      min(floor((X - 28%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size
= 50)!
# ================================================================ #
#def train_net(batch_size=200, learning_rate=1e-4, num_iters=1000, reg
=0.25):


#sweep number of iterations
num_iters = np.linspace(1000, 10000, num=5)
```

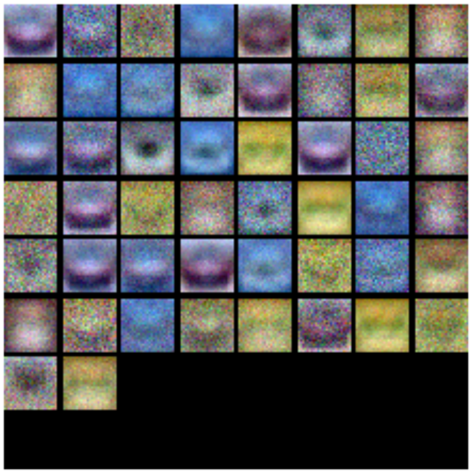
```
regs = np.linspace(0.1, 0.5, num=5)
batch_sizes = np.linspace(100, 800, num=5)
learning_rates = np.linspace(1e-5, 0.002500075, num=5)
#print(learning_rates)

combos = list(itertools.product(num_iters, regs, batch_sizes, learning
_rates))

stats = []
nets = []
val_accs = []
for combo in combos:
    n_iters = int(combo[0])
    reg = combo[1]
    batch_size = int(combo[2])
    lr = combo[3]
#    print("Iters: ", n_iters, " reg: ", reg, " batch_size: ", batch_s
ize, " lr: ", lr)
    val_acc, cur_stats, cur_net = train_net(batch_size=batch_size,
                                            learning_rate=lr,
                                            num_iters=n_iters,
                                           reg=reg)


    stats.append(cur_stats)
    nets.append(cur_net)
    val_accs.append(val_acc)
#    print("val accuracy: ", val_acc)
    if(val_acc > 0.5):
        break

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

```
Validation accuracy:  0.217
Validation accuracy:  0.436
Validation accuracy:  0.446
Validation accuracy:  0.454
Validation accuracy:  0.44
Validation accuracy:  0.204
Validation accuracy:  0.47
Validation accuracy:  0.498
Validation accuracy:  0.495
Validation accuracy:  0.458
Validation accuracy:  0.215
Validation accuracy:  0.466
Validation accuracy:  0.478
Validation accuracy:  0.515
```

In [147]:
```
best_index = np.argmax(val_accs)
best_combo = combos[best_index]
best_net = nets[best_index]
print("First hyperparameter combo over 0.5: \n", combo)
print("Validation accuracy: ", val_accs[best_index])

#generate the average net
val_acc, cur_stats, cur_net = train_net()
subopt_net = cur_net
```

```
First hyperparameter combo over 0.5:
 (1000.0, 0.10000000000000001, 450.0, 0.0018775562500000001)
Validation accuracy:  0.515
Validation accuracy:  0.288
```

In [148]:
```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```



Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

Answer:

(1) The suboptimal net's weights appear to be less pronounced in terms of visual features than the better net. Specific shapes are easily discernable in the better net, whereas the suboptimal net's weights appear to be smoothed or averaged.

Evaluate on test set

In [149]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.497
```

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.

```python
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:

```python
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 [108]:
```
## 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 [109]:
```
# 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 [110]:

```
# 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_shap
e), 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 1e-9.
print('Testing affine_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing affine_forward function:
difference: 9.769850047988 4e-10
```

Affine layer backward pass

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

In [111]:
```
# 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: 1.8445316297490346e-10
dw error: 4.323201630356623e-11
db error: 5.373596178879025e-12
```

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 [112]:
```
# 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,  0.136
36364,],
                        [ 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 [113]:
```
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: 3.2756254812383846e-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 [114]:
```
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: 2.333033877501118e-10
dw error: 1.2885080613487228e-10
db error: 7.826601528069175e-12
```

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 [115]:
```
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, verbo
se=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, v
erbose=False)
loss, dx = softmax_loss(x, y)

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

```
Testing svm_loss:
loss: 9.000807733180942
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.302666299735876
dx error: 1.0307178212406101e-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 [116]:
```
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=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,  14.5719843
4, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.8114912
8, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.0509982
2, 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 l
oss'

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, gra
ds[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 2.131611955458401e-08
W2 relative error: 3.310270199776237e-10
b1 relative error: 8.36819673247588e-09
b2 relative error: 2.530774050159566e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279153413239097e-07
W2 relative error: 2.8508696990815807e-08
b1 relative error: 1.5646802033932055e-08
b2 relative error: 9.089614638133234e-10
```

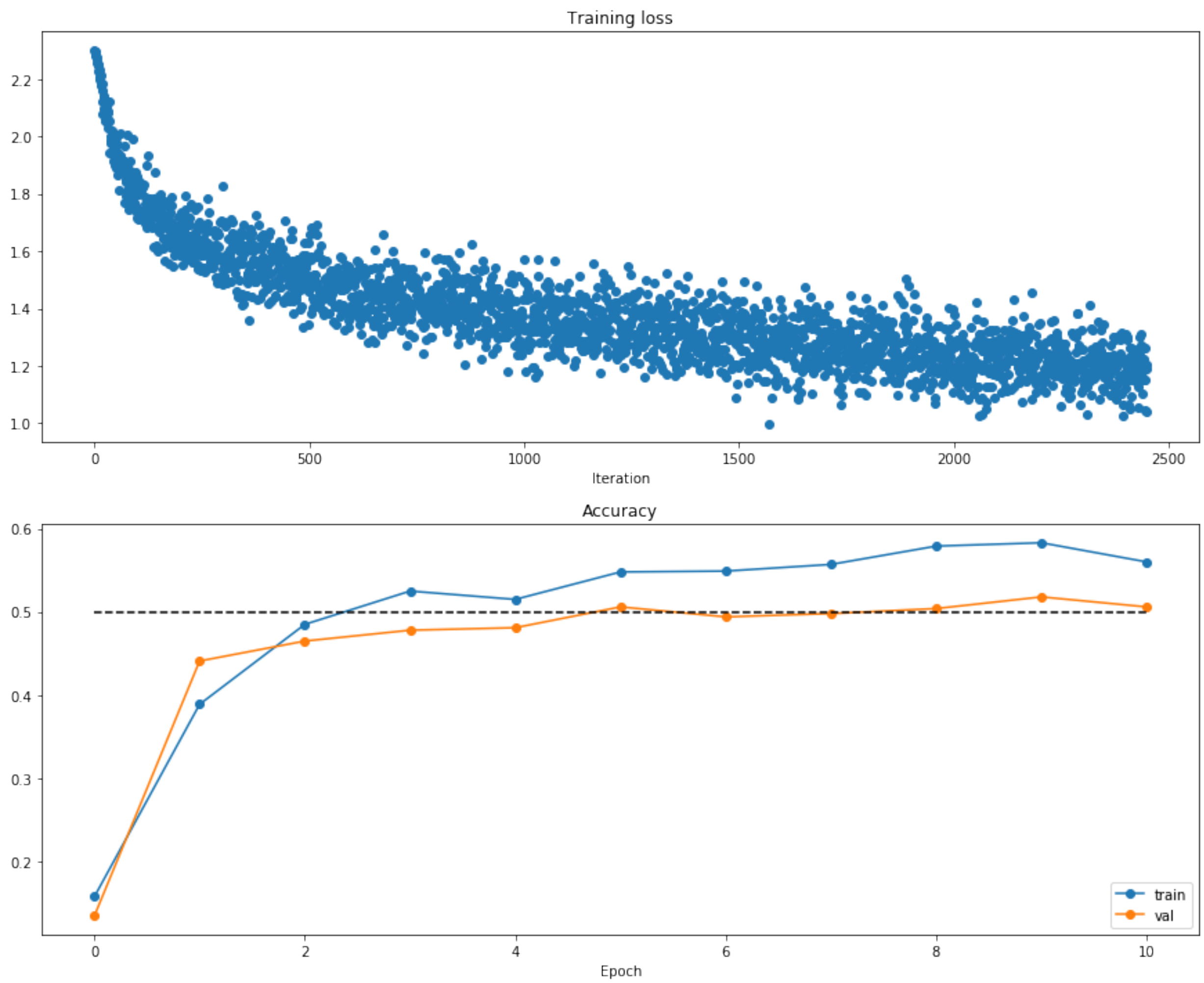
Solver

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 [117]:
```
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 40%.  We won't have you optimize this further
#   since you did it in the previous notebook.
# ================================================================ #

solver = Solver(model, data,
               update_rule='sgd',
               optim_config={'learning_rate':1e-3},
               lr_decay=0.95,
               num_epochs=10,
               batch_size=200,
               print_every=200)
solver.train()

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

```
(Iteration 1 / 2450) loss: 2.301990
(Epoch 0 / 10) train acc: 0.158000; val_acc: 0.135000
(Iteration 201 / 2450) loss: 1.585164
(Epoch 1 / 10) train acc: 0.389000; val_acc: 0.441000
(Iteration 401 / 2450) loss: 1.546105
(Epoch 2 / 10) train acc: 0.485000; val_acc: 0.465000
(Iteration 601 / 2450) loss: 1.515232
(Epoch 3 / 10) train acc: 0.525000; val_acc: 0.478000
(Iteration 801 / 2450) loss: 1.577081
(Epoch 4 / 10) train acc: 0.515000; val_acc: 0.481000
(Iteration 1001 / 2450) loss: 1.573821
(Iteration 1201 / 2450) loss: 1.402064
(Epoch 5 / 10) train acc: 0.548000; val_acc: 0.506000
(Iteration 1401 / 2450) loss: 1.309297
(Epoch 6 / 10) train acc: 0.549000; val_acc: 0.494000
(Iteration 1601 / 2450) loss: 1.403481
(Epoch 7 / 10) train acc: 0.557000; val_acc: 0.498000
(Iteration 1801 / 2450) loss: 1.247962
(Epoch 8 / 10) train acc: 0.579000; val_acc: 0.504000
(Iteration 2001 / 2450) loss: 1.149341
(Iteration 2201 / 2450) loss: 1.166358
(Epoch 9 / 10) train acc: 0.583000; val_acc: 0.518000
(Iteration 2401 / 2450) loss: 1.309006
(Epoch 10 / 10) train acc: 0.560000; val_acc: 0.506000
```

In [118]:
```
# 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.

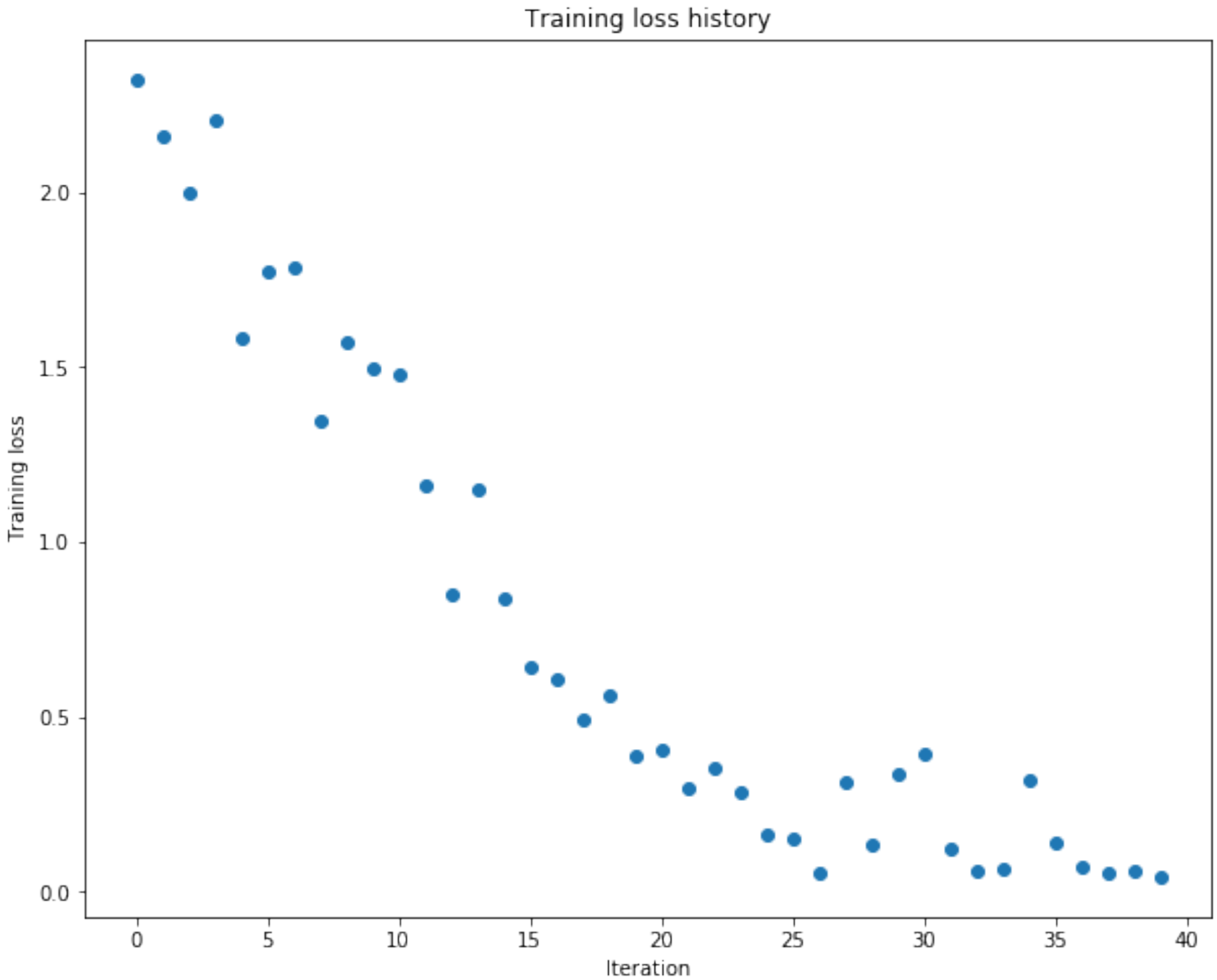
In [119]:

```
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.float
64)

  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, gra
ds[name])))
```

```
Running check with reg = 0
Initial loss: 2.3042769950299
W1 relative error: 5.78358042694402e-07
W2 relative error: 6.42570208842477e-07
W3 relative error: 4.657381714075064e-08
b1 relative error: 1.1120627057963377e-08
b2 relative error: 9.089910601530006e-10
b3 relative error: 1.0152776296277876e-10
Running check with reg = 3.14
Initial loss: 6.746964977126233
W1 relative error: 3.005955787243122e-08
W2 relative error: 1.042974285738492e-07
W3 relative error: 5.417855468854517e-09
b1 relative error: 1.73428594392816e-08
b2 relative error: 1.3920951552600733e-08
b3 relative error: 2.0054775589097045e-10
```

In [120]:
```
# 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 = 1e-2
learning_rate = 1e-2

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: 2.324044
(Epoch 0 / 20) train acc: 0.300000; val_acc: 0.126000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.157000
(Epoch 2 / 20) train acc: 0.400000; val_acc: 0.148000
(Epoch 3 / 20) train acc: 0.480000; val_acc: 0.141000
(Epoch 4 / 20) train acc: 0.420000; val_acc: 0.162000
(Epoch 5 / 20) train acc: 0.660000; val_acc: 0.185000
(Iteration 11 / 40) loss: 1.481542
(Epoch 6 / 20) train acc: 0.640000; val_acc: 0.160000
(Epoch 7 / 20) train acc: 0.800000; val_acc: 0.191000
(Epoch 8 / 20) train acc: 0.920000; val_acc: 0.206000
(Epoch 9 / 20) train acc: 0.920000; val_acc: 0.215000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.201000
(Iteration 21 / 40) loss: 0.402930
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.175000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.186000
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.182000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.192000
(Epoch 15 / 20) train acc: 0.960000; val_acc: 0.175000
(Iteration 31 / 40) loss: 0.392616
(Epoch 16 / 20) train acc: 0.960000; val_acc: 0.199000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.213000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.199000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.201000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.201000
```


Training loss history
Training loss
Iteration
0.0
0.5
1.0
1.5
2.0
0
5
10
15
20
25
30
35
40

neural_net.py

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

"""
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 t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
cs231n.stanford.edu.
"""

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dim
ension of
  N, a hidden layer dimension of H, and performs classification over C
classes.
  We train the network with a softmax loss function and L2 regularizat
ion on the
  weight matrices. The network uses a ReLU nonlinearity after the firs
t fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softm
ax

  The outputs of the second fully-connected layer are the scores for e
ach class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random valu
es and
    biases are initialized to zero. Weights and biases are stored in t
he
    variable self.params, which is a dictionary with the following key
s:
```

```
    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size
)
    self.params['b2'] = np.zeros(output_size)


  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neu
ral
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and ea
ch y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optiona
l; if it
      is not passed then we only return scores, and if it is passed th
en we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[
i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of
training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those
parameters
      with respect to the loss function; has the same keys as self.par
ams.
    """
    # Unpack variables from the params dictionary
```

```
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the output scores of the neural network.  The result
    #   should be (C, N). As stated in the description for this class,
    #   there should not be a ReLU layer after the second FC layer.
    #   The output of the second FC layer is the output scores. Do not
    #   use a for loop in your implementation.
    # ================================================================
#
#    print(N,D)
    #  input - fully connected layer - ReLU - fully connected layer -
softmax
    #first layer
    HL1_pre_activation = X.dot(W1.T) + b1
    HL1_output = np.maximum(0, HL1_pre_activation) #relu

    #second layer
    HL2_pre_activation = HL1_output.dot(W2.T) + b2

    scores = HL2_pre_activation

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


    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = 0.0

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the loss of the neural network.  This includes the
    #   softmax loss and the L2 regularization for W1 and W2. Store th
e
    #   total loss in the variable loss.  Multiply the regularization
```

```
    #   loss by 0.5 (in addition to the factor reg).
    # ==================================================================
#

    # scores is num_examples by num_classes
    #Loss is made up of standard softmax loss and L2 regularization
    #Generate probability of being in a class based on output (softmax
)
    class_probabilities = np.exp(scores)/np.sum(np.exp(scores), axis=1
, keepdims=True)

    """
    There will be N rows, where each row corresponds to an input.
    There are D columns, where each column will correspond to probabil
ity of being in that class.
    y is our gnd truth, so for some y=j and example i, we want class_p
robabilities[i, y=j]
    """
#    print(y)
 #   print(class_probabilities)
#    print(range[N])
    prob_of_correct_y = class_probabilities[np.arange(N), y]
    log_loss = -np.log(prob_of_correct_y)
    sum_log_loss = np.sum(log_loss)
    #divide by num examples
    loss = sum_log_loss/N

    """
    L2 regularization for matrix involves Frobenius norm.
    reg = 0.5*|| w ||_F ^2
    Frobenius norm is equiv to Sigma_iSigma_j(w_ij)^2, so we can just
do a dual sum
    """
    frob_norm_w1 = np.sum(W1**2)
    frob_norm_w2 = np.sum(W2**2)
    reg_w1 = 0.5*reg*frob_norm_w1
    reg_w2 = 0.5*reg*frob_norm_w2

    regularized_loss = reg_w1 + reg_w2
    loss += regularized_loss


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

    grads = {}
```

```
    # ===========================================================================
#
    # YOUR CODE HERE:
    #   Implement the backward pass.  Compute the derivatives of the
    #   weights and the biases.  Store the results in the grads
    #   dictionary.  e.g., grads['W1'] should store the gradient for
    #   W1, and be of the same size as W1.
    # ===========================================================================
#
    """
    Source: CS231n online
    Gradient of L_i = -log(p_yi) is p_k-1 for (y_i = k)

    For weights we do a mult between the previous layer output and the
update

    We will multiply by the negative learning rate so a weight "decrea
se" at an intermediate step
    is really a weight increase.
    """



    #Calculate how we should update the scores
    update_scores = class_probabilities
    #Since we made update scores matrix by looking for only cases wher
e y_i = k, we can subtract
    #from the whole thing
#    update_scores -= np.ones_like(update_scores)
    update_scores[np.arange(N), y] -=1
    update_scores /= N

#    print(update_scores)
    #backprop W2 take gradient of output and multiply by weight matirx
    grads['W2'] = np.dot(HL1_output.T, update_scores).T

    #we want to increase the value of the activation of correct classi
fications
    grads['b2'] = np.sum(update_scores, axis=0)#, keepdims=True)

    # dL/dW2 = dL/dOut * dOut/dW2
    dHL2 = np.dot(update_scores, W2)

    # I(a>0)*dl/dh (where h is output of relu layer)
    # a in this case is HL1_pre_activation
    dLdA = dHL2
    dLdA[HL1_output <= 0] = 0


    #back prop DlDa into w and b
    grads['W1'] = np.dot(dLdA.T, X)
```

```
    grads['b1'] = np.sum(dLdA, axis=0)#, keepdims=True)


    grads['W2'] += reg * W2
    grads['W1'] += reg * W1




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

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c m
eans that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the lear
ning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in np.arange(num_iters):
      X_batch = None
```

```
        y_batch = None

        # ================================================================
== #
        # YOUR CODE HERE:
        #   Create a minibatch by sampling batch_size samples randomly.
        # ================================================================
== #
        rand_indices = np.random.choice(np.arange(num_train), batch_size
)
        X_batch = X[rand_indices]
        y_batch = y[rand_indices]
        # ================================================================
== #
        # END YOUR CODE HERE
        # ================================================================
== #

         # Compute loss and gradients using the current minibatch
        loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
        loss_history.append(loss)

        # ================================================================
== #
        # YOUR CODE HERE:
        #   Perform a gradient descent step using the minibatch to updat
e
        #   all parameters (i.e., W1, W2, b1, and b2).
        # ================================================================
== #

        self.params['W2'] += -learning_rate * grads['W2']
        self.params['W1'] += -learning_rate * grads['W1']

#        print(self.params['b2'].shape, grads['b2'].shape)
        self.params['b2'] += -learning_rate * grads['b2']
        self.params['b1'] += -learning_rate * grads['b1']

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

        if verbose and it % 100 == 0:
          print('iteration {} / {}: loss {}'.format(it, num_iters, loss)
)

        # Every epoch, check train and val accuracy and decay learning r
ate.
```

```
      if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }

  def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict label
s for
    data points. For each data point we predict scores for each of the
C
    classes, and assign each data point to the class with the highest
score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data poi
nts to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for
each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is p
redicted
      to have class c, where 0 <= c < C.

    W1: First layer weights; has shape (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    b2: Second layer biases; has shape (C,)
    """
    num_examples = X.shape[0]
    y_pred = np.empty((num_examples,), dtype=int)

    # ================================================================
#
    # YOUR CODE HERE:
    #   Predict the class given the input data.
    # ================================================================
```

```
#
    #do a forward pass for prediction
    HL1_input = np.dot(X, self.params['W1'].T) + self.params['b1']

    #apply RELU
    HL1_output = np.maximum(0, HL1_input)

    #second layer
    HL2_output = np.dot(HL1_output, self.params['W2'].T) + self.params
['b2']

    #apply softmax
    softmax = np.exp(HL2_output)/np.sum(np.exp(HL2_output), axis=1, ke
epdims=True)

    #index_of_max = np.argmax(softmax)
#     print(softmax.shape)
    for i in range(num_examples):
      max_index = np.argmax(softmax[i])
      y_pred[i] = max_index




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

    return y_pred
```

layers.py

In [1]:
```
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 t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
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, a
nd
  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.
  # ================================================================ #
  N = x.shape[0]
```

```
  D = w.shape[0]
  x_reshaped = np.reshape(x, (N,D))
  out = x_reshaped.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.
  # ================================================================== #

  #reshape x matrix to be N, D and multiply upstream for the chain rul
e
  N = x.shape[0]
  D = w.shape[0]
  reshaped_x = np.reshape(x, (N, D))
  dw = reshaped_x.T.dot(dout)

  #derivative wrt x
  dx_raw = dout.dot(w.T)
  dx = np.reshape(dx_raw, x.shape)
  #sum derivative for bias
  db = np.sum(dout, axis=0)
```

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

    return dx, dw, db

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

    Input:
    - x: Inputs, of any shape

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

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

    cache = x
    return out, cache


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

    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 backward pass multiplies the dout by the indicator function
  #arr[arr > 255] = x
  dx = dout

  #apply indicator. Uses < and not <= because 0 is undefined for ReLU
  dx[x < 0] = 0
  # ======================================================================== #
  # END YOUR CODE HERE
  # ======================================================================== #

  return dx

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

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

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


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

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

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

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

fc_net.py

In [1]:
```
import numpy as np

from .layers import *
from .layer_utils import *

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

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

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

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

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

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

    Inputs:
```

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

    # ===================================================================
#
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1
'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. Th
e
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weigh
t_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ===================================================================
#
    mu = 0
    sigma = weight_scale
    self.params['W1'] = np.random.normal(mu, sigma, (input_dim, hidden
_dims))
    self.params['W2'] = np.random.normal(mu, sigma, (hidden_dims, num_
classes))


    self.params['b1'] = np.zeros(shape = (hidden_dims))
    self.params['b2'] = np.zeros(shape = (num_classes))

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

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

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

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

    If y is not None, then run a training-time forward and backward pa
ss and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping par
ameter
      names to gradients of the loss with respect to those parameters.
    """
     # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']


    # Compute the forward pass
    scores = None
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. St
ore
    #   the class scores as the variable 'scores'.  Be sure to use the
layers
    #   you prior implemented.
    # ================================================================
#

    #affine_relu_forward returns out, cached
    hl1, hl1_cached = affine_relu_forward(X, W1, b1)

    #affine_forward returns out, cache
    hl2, hl2_cached = affine_forward(hl1, W2, b2)

    scores = hl2
    scores_cached= hl2_cached
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
```

```
    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss, grads = 0, {}

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Stor
e
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] hol
ds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, et
c.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.r
eg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ================================================================
#

    """
    Softmax loss returns:
     Returns a tuple of:
      - loss: Scalar giving the loss
      - dx: Gradient of the loss with respect to x
  """
    loss, grad_loss_wrt_x = softmax_loss(scores, y)
    regularized_loss = 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.su
m(W2**2)
    loss += regularized_loss

    #backprop
    #affine return values:  return dx, dw, db
    dx2, dw2, db2 = affine_backward(grad_loss_wrt_x, scores_cached)

    #add in regularization term to weights
    dw2 += self.reg*W2

    #backprop layer 1
    #relu_backward takes: (dout, cache). In this case the dout is prev
ious chain
    dx1, dw1, db1 = affine_relu_backward(dx2, hl1_cached)
```

```
    dw1 += self.reg*W1

    grads['W2'] = dw2
    grads['b2'] = db2
    grads['W1'] = dw1
    grads['b1'] = db1

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

    return loss, grads


class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden
layers,
  ReLU nonlinearities, and a softmax loss function. This will also imp
lement
  dropout and batch normalization as options. For a network with L lay
ers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - soft
max

  where batch normalization and dropout are optional, and the {...} bl
ock is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in
the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden l
ayer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify
.
```

```
    - dropout: Scalar between 0 and 1 giving dropout strength. If drop
out=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch norma
lization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be perform
ed using
      this datatype. float32 is faster but less accurate, so you shoul
d use
      float64 for numeric gradient checking.
    - seed: If not None, then pass this random seed to the dropout lay
ers. This
      will make the dropout layers deteriminstic so we can gradient ch
eck the
      model.
    """
    self.use_batchnorm = use_batchnorm
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len(hidden_dims)
    self.dtype = dtype
    self.params = {}

    # ================================================================
#
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params di
ctionary.
    #   The weights and biases of layer 1 are W1 and b1; and in genera
l the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weigh
t_scale.
    # ===========================================e====================
= #
    mu = 0
    stddev = weight_scale
    """
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size
)
    self.params['b2'] = np.zeros(output_size)

    np.random.normal(mu, stddev, <size>)
    """
```

```
    #aggregate all the dims into a single array that we can reference
    #input and output dim (num_classes) will only be used once
    aggregated_dims = [input_dim] + hidden_dims + [num_classes]
    for i in range(self.num_layers):
      self.params['b'+str(i+1)] = np.zeros(aggregated_dims[i+1])
      self.params['W'+str(i+1)] = np.random.normal(mu, stddev, size=(a
ggregated_dims[i], aggregated_dims[i+1]))
    # ===================================================================
#
    # END YOUR CODE HERE
    # ===================================================================
#

    # When using dropout we need to pass a dropout_param dictionary to
each
    # dropout layer so that the layer knows the dropout probability an
d the mode
    # (train / test). You can pass the same dropout_param to each drop
out layer.
    self.dropout_param = {}
    if self.use_dropout:
      self.dropout_param = {'mode': 'train', 'p': dropout}
      if seed is not None:
        self.dropout_param['seed'] = seed

    # With batch normalization we need to keep track of running means
and
    # variances, so we need to pass a special bn_param object to each
batch
    # normalization layer. You should pass self.bn_params[0] to the fo
rward pass
    # of the first batch normalization layer, self.bn_params[1] to the
forward
    # pass of the second batch normalization layer, etc.
    self.bn_params = []
    if self.use_batchnorm:
      self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_
layers - 1)]

    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
```

```
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since
they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ==================================================================
#
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    # ==================================================================
#

    nn_layer = {}
    nn_cache = {}

    #initialize the first layer with the inputs
    nn_layer[0] = X
    #pass through each layer
    for i in range(1, self.num_layers):
      #affine relu forward takes (x, w, b)
      nn_layer[i], nn_cache[i] = affine_relu_forward(nn_layer[i-1], se
lf.params['W'+str(i)], self.params['b'+str(i)])

    #all layers will have the affine_relu except for the last layer, w
hich is a passthrough
    #affine_forward takes (x, w, b) and outputs out, cache
    w_idx = 'W'+str(self.num_layers)
    b_idx = 'b'+str(self.num_layers)
    scores, cached_scores = affine_forward(nn_layer[self.num_layers -1
], self.params[w_idx], self.params[b_idx])


    # ==================================================================
#
    # 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 gradi
ents
    #   in the grads dict, so that grads[k] is the gradient of self.pa
rams[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    # ============================================================
#
    #get loss w/ softmax loss
    loss, grad_loss = softmax_loss(scores, y)

    #add L2 regularization to loss 1/2*np.sum(w**2)
    for i in range(1, self.num_layers + 1):
      cur_weight_matrix = self.params['W'+str(i)]
      loss += 0.5 * self.reg * np.sum(cur_weight_matrix**2)

    """
    Backpropping into the (n-1)th layer will be different because we d
on't have
    the relu. Use affine_backward and then for each previous layer app
ly affine_relu_backward
    affine_backward takes dout,cache and returns dx,dw,db
    affine_relu_backward takes dout, cachce and returns dx,dw,db
    """
    dx={}
    w_idx_nth = 'W'+str(self.num_layers)
    b_idx_nth = 'b'+str(self.num_layers)
    dx[self.num_layers], grads[w_idx_nth], grads[b_idx_nth] = affine_b
ackward(grad_loss, cached_scores)

    #regularize
    grads[w_idx_nth] += self.reg * self.params[w_idx_nth]

    #we apply affine_relu_backward now
    for i in range(self.num_layers - 1, 0, -1):
#      print(i, self.num_layers)

      #dx, dw, db
      w_idx = 'W' + str(i)
      b_idx = 'b' + str(i)

      #dout input to affine_relu_backward is the
      dx[i], grads[w_idx], grads[b_idx] = affine_relu_backward( dx[i+1
], nn_cache[i])
```

```
        #regularize
        grads[w_idx] += self.reg * self.params[w_idx]


    # ==================================================================
#
    # END YOUR CODE HERE
    # ==================================================================
#
    return loss, grads
```