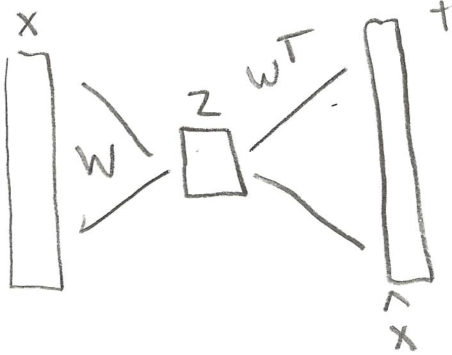
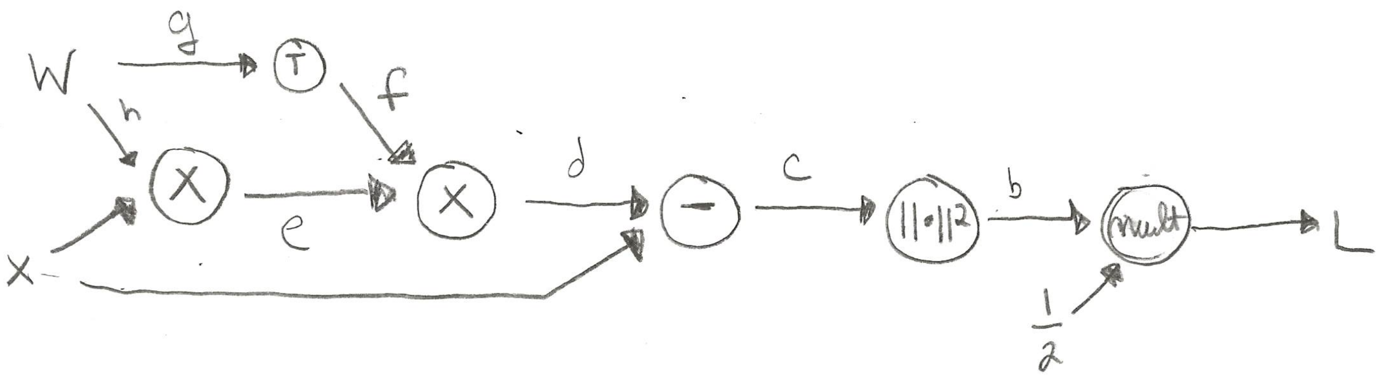
1.

a. The matrix multiplication of Wx projects x into a latent dimension z, and then the matrix multiplication of W^Tz tries to reconstruct the original projected x from z. Thus the l_2 norm penalizes entries of the reconstructed x, $\hat{x}$, that does not match X. To reconstruct well the projected x (z) must also be optimized to highly represent x.



$\hat{x}-x$ should be close, and if this does not hold we penalize! $\hat{x}=W^Tz$.

b.



c. To account for these two paths to W when calculating $\nabla_W C$, you add the gradients that occur from the two paths. I.e. The law of total derivatives

$$\nabla_W C = \nabla_g L + \nabla_h L$$

$$= \frac{dWx}{dW}\frac{dL}{dWx} + \frac{dW^T}{dW}\frac{dL}{dW^T}$$

d.

$L = \frac{1}{2} b$

Scalar $b = \|c\|^2 = c^T c$

$n \times 1$ $c = d - x$

$n \times 1$ $d = fe$

$m \times 1$ $e = hx$

$n \times m$ $f = g^T$

$m \times n$ $h = W$

$m \times n$ $g = W$

$$\frac{dL}{db} = \frac{1}{2}$$

$$\overset{n\times 1}{\frac{dL}{dc}} = \overset{n\times 1}{\frac{db}{dc}} \cdot \overset{1\times 1}{\frac{dL}{db}} = 2c \cdot \frac{1}{2} = c$$

$$\overset{n\times 1}{\frac{dL}{dd}} = \overset{n\times n}{\frac{dc}{dd}} \cdot \overset{n\times 1}{\frac{dL}{dc}} = I \cdot c = c$$

$$\overset{n\times m}{\frac{dL}{df}} = \overset{n\times m\times n}{\frac{dd}{df}} \cdot \overset{n\times 1}{\frac{dL}{dd}} = \overset{n\times 1}{\frac{dL}{dd}} \overset{1\times m}{e^T}$$ (trick from formal notes)

$$\frac{d d_1}{df} = \begin{bmatrix} - e^T - \\ 0 \\ | \end{bmatrix} \quad \frac{d d_2}{df} = \begin{bmatrix} 0 \\ - e^T - \\ 0 \\ | \end{bmatrix} \quad \frac{d d_n}{df} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ - e^T - \end{bmatrix}$$

$$\overset{m\times 1}{\frac{dL}{de}} = \overset{m\times n}{\frac{dd}{de}} \cdot \overset{n\times 1}{\frac{dL}{dd}} = \overset{m\times n}{f^T} \overset{n\times 1}{c}$$

$$\overset{m\times n}{\frac{dL}{dg}} = \overset{(m\times n)\times(n\times m)}{\frac{df}{dg}} \cdot \overset{(n\times m)\times 1}{\frac{dL}{df}}$$

$$= \overset{m\times n}{\left(\frac{dL}{df}\right)^T}$$

$$\frac{df_{1,1}}{dg} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \ddots & & \\ 0 & & & \\ 0 & & & \end{bmatrix} \quad \frac{df_{1,2}}{dg} = \begin{bmatrix} 0 & - & - \\ 1 & \ddots & \\ 0 & & \\ | & & \end{bmatrix} \quad \frac{df_{2,1}}{dg} = \begin{bmatrix} 0 & 1 & 0 & 0 & - \\ 0 & \ddots & & & \\ | & & & & \end{bmatrix}$$

$$\frac{df_{n,m}}{dg} = \begin{bmatrix} 0 & 0 & - \\ 0 & \ddots & \\ | & & 1 \end{bmatrix} \quad \therefore \frac{dL}{dg} = \left(\frac{dL}{df}\right)_{1,1} \begin{bmatrix} 1 & 0 & - \\ 0 & \ddots & \\ | & & \end{bmatrix} + \cdots + \left(\frac{dL}{df}\right)_{n,m} \begin{bmatrix} 0 & - \\ | & \ddots \\ & & 1 \end{bmatrix}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{m} \left(\frac{dL}{df}\right)_{i,j} \left(\frac{df_{i,j}}{dg}\right)$$

$$\frac{dL}{dh} = \overset{m\times n\times m}{\frac{de}{dh}} \cdot \overset{m\times 1}{\frac{dL}{de}} = \overset{m\times 1}{\frac{dL}{de}} \overset{1\times n}{x^T}$$

(same trick from formal notes)

$$\frac{\partial L}{dw} = \frac{dL}{\partial g} + \frac{dl}{dh}$$

$$\frac{dL}{\partial g} = \left(\frac{\partial L}{\partial f}\right)^T = \left(\frac{\partial L}{\partial d} e^T\right)^T = \left(c e^T\right)^T = \left((d-x)e^T\right)^T = \left((fe-x)e^T\right)^T = \left((w^T e - x)e^T\right)^T$$

$$= \left((w^T W x - x)(Wx)^T\right)^T = Wx\,(\overbrace{W^T W x - x}^{1\times n})^T$$

$n\times 1$ $n\times 1$

$m\times n$ $n\times 1$

$(m\times 1) \cdot (1\times n) = m\times n$

$$\frac{\partial L}{dh} = \frac{\partial L}{\partial c} x^T = (f^T c)x^T = (f^T(d-x))x^T = (f^T(w^T e - x))x^T = (W(w^T Wx - x))x^T$$

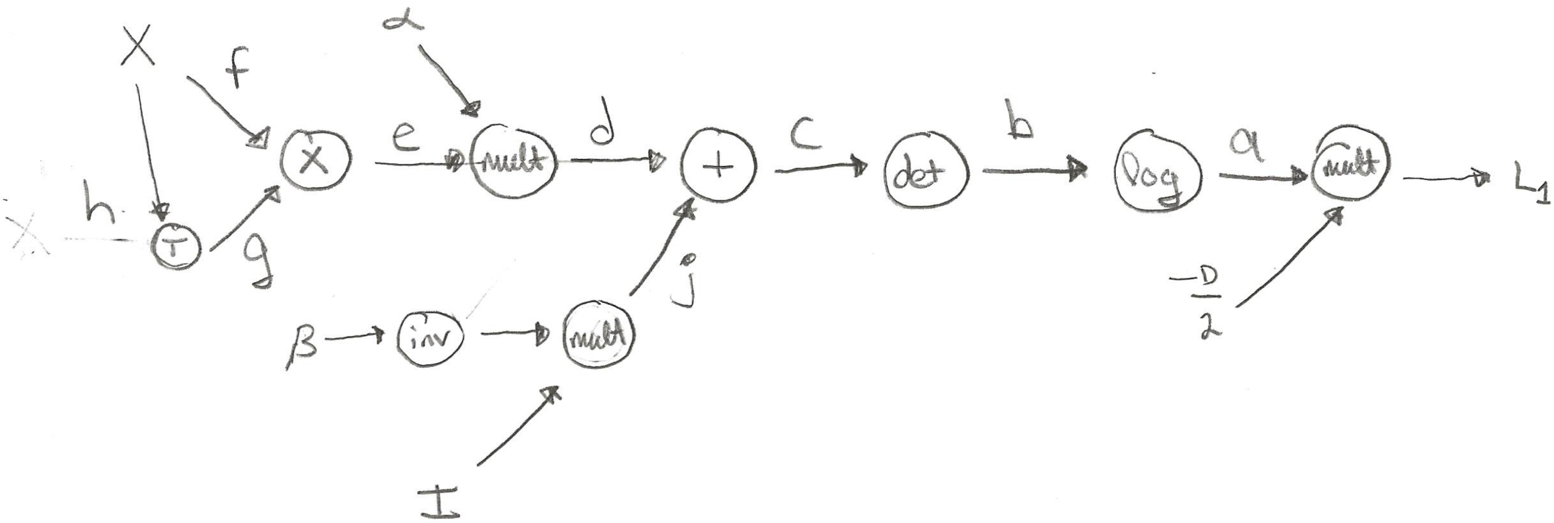
$$= (\underbrace{W}_{m\times n}\underbrace{(W^T Wx - x)}_{1\times 1})\underbrace{x^T}_{1\times n}$$

$= m\times n$

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

2.

a.



(b)

scalar

$L_1 = \frac{-D}{2} \cdot a$ | scalar $\frac{dL}{da} = \frac{-D}{2}$

scalar 1x1 $a = \log b$ | scalar $\frac{dL}{db} = \frac{da}{db} \cdot \frac{dL}{da} = \frac{1}{b} \cdot \left(\frac{-D}{2}\right) = \frac{-D}{2b}$

scalar 1x1 $b = \det c$ | 2D matrix $\underset{m\times m}{\frac{dL}{dc}} = \underset{m\times m}{\frac{db}{dc}} \cdot \underset{1\times 1}{\frac{dL}{db}} = \underbrace{\det(c)(c^{-1})^T}_{\text{matrix (from cookbook)}} \cdot \underbrace{\frac{dL}{db}}_{\text{scalar}}$

matrix mxm $C = j + d$

matrix mxm $d = \alpha e$ | 2D matrix $\frac{dL}{dd} = \frac{dc}{dd} \cdot \frac{dL}{dc} = \left(\frac{dL}{dc}\right)$ (showed derivations in last problem)

mxm; 4D matrix mxm x (mxm); 2D matrix (mxm) x 1

matrix mxm $e = f g$

matrix nxm $g = h^T$ | 2D matrix mxm $\frac{dL}{de} = \frac{dd}{de} \cdot \frac{dL}{dd} = \alpha\left(\frac{dL}{dd}\right)$ (showed derivations in last proble

4D matrix mxmxmxm; 2D matrix (mxm)x1

matrix mxn $h = X$

$f = X$

$X \in m\times n$

$I \in m\times m$

2D matrix $\frac{dL}{dg} = \frac{de}{dg} \cdot \frac{dL}{de} = f^T \frac{dL}{de}$

nxm; 4D matrix nxmxmxm; 2D matrix mxm; mxm; mxm

$$\frac{\partial L}{\partial h} = \frac{\partial g}{\partial h} \circ \frac{\partial L}{\partial g} = \left(\frac{\partial L}{\partial g}\right)^T \rightarrow \text{trick from before}$$

$m \times n \quad m \times n \times n \times m \quad n \times m$

$$\frac{\partial L}{\partial f} = \frac{\partial e}{\partial f} \cdot \frac{\partial L}{\partial e} = \frac{\partial L}{\partial e} \circ g^T$$

$m \times n \quad m \times n \times m \times m \quad m \times m \qquad m \times m \quad m \times n$

$$\frac{\partial L_1}{\partial X} = \frac{\partial L}{\partial f} + \frac{\partial L}{h}$$

$$\frac{\partial L}{\partial f} = \frac{\partial L}{\partial e} \cdot g^T = \alpha \frac{\partial L}{\partial d} \cdot g^T = \alpha \frac{\partial L}{\partial c} g^T = \alpha \det(c)(c^{-1})^T \circ \frac{\partial L}{\partial b} \circ g^T$$

$$= \alpha \det(c)(c^{-1})^T \circ \left(\frac{-p}{2b}\right) \circ g^T$$

$$= \alpha \det(j+d)((j+d)^{-1})^T \circ \frac{-D}{2\det c} \circ X$$

$$= \underbrace{\left(\alpha \det\left(\tfrac{1}{\beta}I + \alpha XX^T\right)\underbrace{\left(\left(\tfrac{1}{\beta}I + \alpha XX^T\right)^{-1}\right)^T}_{m \times m} \circ \frac{-P}{2\det\left(\tfrac{1}{\beta}I + \alpha XX^T\right)}\right)}_{m \times n} \circ X$$

$$\frac{\partial L}{\partial h} = \left(\frac{\partial L}{\partial g}\right)^T = \frac{\partial L}{\partial e}^T f = \left(\frac{-D\alpha}{2\det\left(\tfrac{1}{\beta}I + \alpha XX^T\right)} \det\left(\tfrac{1}{\beta}I + \alpha XX^T\right)\left(\left(\tfrac{1}{\beta}I + \alpha XX^T\right)^{-1}\right)^T\right)^T X$$

$$K = \alpha X X^T + \beta^{-1} I$$

$$\frac{dL_1}{dX} = \frac{dL_1}{dh} + \frac{dL_1}{df}$$

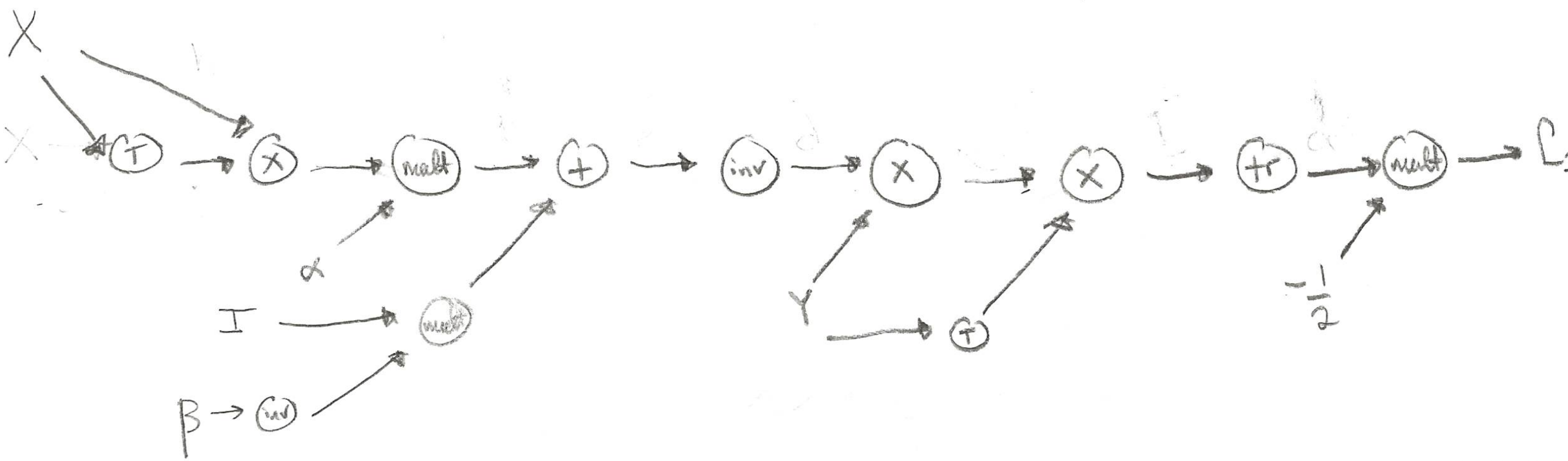
$$= \alpha \det(K)(K^{-1})^T \left(\frac{-D}{2\det(K)}\right) \cdot X + \left(\frac{-D\alpha}{2\det(K)} \det(K)\,((K^{-1})^T)\right)^T X$$

$$\frac{-D\alpha}{2}(K^{-1})^T X + \frac{-D\alpha}{2}(K^{-1}) X$$

$$\boxed{\frac{dL_1}{dX} = \frac{-D\alpha}{2}\left[(K^{-1})^T + (K^{-1})\right] X = -D\alpha K^{-1} X}$$

$$K^T = K$$

(c)



d. $\frac{\partial L}{\partial K} = -K^{-T} \frac{\partial L}{\partial K^{-1}} K^{-T}$

$\frac{\partial Tr(XA)}{\partial X} = X^T$ $\frac{\partial Tr(K^{-1}YY^T{}_m)}{\partial K^{-1}} = YY^T$

$$\frac{\partial L}{\partial X} = \underbrace{\frac{\partial K}{\partial X}}_{m \times n \times m \times m} \underbrace{\frac{\partial L}{\partial K}}_{m \times m} = \boxed{\alpha K^{-T} YY^T K^{-T} X = \frac{\partial L_2}{\partial X}}.$$

$$\frac{\partial K}{\partial X}\frac{\partial L}{\partial K} = \sum_{ij} \left(\frac{\partial K_{ij}}{\partial X}\right)\left(\frac{\partial L}{\partial K}\right)_{ij}$$

if $\frac{\partial L}{\partial K}$ is symmetric... train of thought

$$\underbrace{\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} x_1 & x_3 \\ x_2 & x_4 \end{bmatrix}}_{X^T} = \begin{bmatrix} x_1^2 + x_2^2 & x_1x_3 + x_2x_4 \\ x_1x_3 + x_2x_4 & x_3^2 + x_4^2 \end{bmatrix} \qquad \begin{bmatrix} a & b \\ b & d \end{bmatrix}$$

one 2x2 slice is $\frac{\partial K_{ij}}{\partial X}$

$$\begin{bmatrix} 2x_1 & 2x_2 \\ 0 & 0 \end{bmatrix} a + \begin{bmatrix} x_3 & x_4 \\ x_1 & x_2 \end{bmatrix} b + \begin{bmatrix} x_3 & x_4 \\ x_1 & x_2 \end{bmatrix} b + \begin{bmatrix} 0 & 0 \\ 2x_3 & 2x_4 \end{bmatrix} d$$

$$= \begin{bmatrix} 2ax_1 + 2x_3b & 2ax_2 + 2bx_4 \\ 2x_3d + 2x_1b & 2x_4d + 2x_2b \end{bmatrix} = 2\begin{bmatrix} a & b \\ b & d \end{bmatrix}\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$$

e.

$$L = -c - \frac{D}{2}\log|K| - \frac{1}{2}\mathrm{tr}(K^{-1}YY^T)$$

$$\frac{\partial L}{\partial X} = \frac{\partial L_1}{\partial X} + \frac{\partial L_2}{\partial X}$$

$$\frac{\partial L}{\partial X} = \frac{-D\alpha}{2}\left[(K^{-1})^T + (K^{-1})\right]X + \alpha K^{-T}YY^TK^{-T}X$$

$$= \alpha\left[\frac{-D}{2}\left[(K^{-T}) + K^{-1}\right] + K^{-T}YY^TK^{-T}\right]X$$

$$= \alpha\left[DK^{-1} + K^{-1}YY^TK^{-1}\right]X$$

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 [1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_grad
ient_array
from cs231n.solver import Solver

import os
#alias kk os._exit(0)

%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))))
```

In [2]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()

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

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

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 [3]:
```
# 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: 9.769847728806635e-10
```

Affine layer backward pass

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

In [5]:
```
# 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
.reshape(10,-6), 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: 2.7825711313218806e-09
dw error: 6.478284543571568e-09
db error: 9.272076782582778e-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 [6]:
```
# 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 [7]:
```
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: 3.2756388851377516e-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 [8]:
```
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.reshape(2,12), 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: 8.196932245092791e-10
dw error: 7.335977111233327e-09
db error: 1.892886971142364e-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 [9]:

```
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: 9.001020468684722
dx error: 1.4021566006651672e-09

Testing softmax_loss:
loss: 2.3026875685808417
dx error: 1.0021775812402786e-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 [10]:
```
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: 1.52157032098804e-08
W2 relative error: 3.4803693682531243e-10
b1 relative error: 6.5485462766289595e-09
b2 relative error: 4.3291413857436005e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 8.175466255230509e-07
W2 relative error: 2.8508696990815807e-08
b1 relative error: 1.0895969390651956e-09
b2 relative error: 9.089615724390711e-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 40%.

In [11]:
```
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,)
```

In [12]:
```
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.
#
# ================================================================ #
N, D, H, C = 49000, 3*32*32, 50, 10
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)

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


solver.train()
pass

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

```
(Iteration 1 / 4900) loss: 3.290638
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.123000
(Iteration 11 / 4900) loss: 2.389829
(Iteration 21 / 4900) loss: 2.215848
(Iteration 31 / 4900) loss: 2.212143
(Iteration 41 / 4900) loss: 1.994485
(Iteration 51 / 4900) loss: 2.223893
(Iteration 61 / 4900) loss: 1.950839
(Iteration 71 / 4900) loss: 1.929391
(Iteration 81 / 4900) loss: 2.165801
(Iteration 91 / 4900) loss: 1.996686
(Iteration 101 / 4900) loss: 1.819187
(Iteration 111 / 4900) loss: 1.795328
(Iteration 121 / 4900) loss: 1.822808
(Iteration 131 / 4900) loss: 1.744277
(Iteration 141 / 4900) loss: 1.886829
(Iteration 151 / 4900) loss: 1.699980
(Iteration 161 / 4900) loss: 1.901897
(Iteration 171 / 4900) loss: 1.735307
(Iteration 181 / 4900) loss: 1.768070
(Iteration 191 / 4900) loss: 1.764556
(Iteration 201 / 4900) loss: 1.852760
(Iteration 211 / 4900) loss: 1.652856
(Iteration 221 / 4900) loss: 1.740637
(Iteration 231 / 4900) loss: 1.769938
(Iteration 241 / 4900) loss: 1.709421
(Iteration 251 / 4900) loss: 1.792931
(Iteration 261 / 4900) loss: 1.474497
(Iteration 271 / 4900) loss: 1.785949
(Iteration 281 / 4900) loss: 1.700124
(Iteration 291 / 4900) loss: 1.679014
(Iteration 301 / 4900) loss: 1.834070
(Iteration 311 / 4900) loss: 1.796900
(Iteration 321 / 4900) loss: 1.656792
(Iteration 331 / 4900) loss: 1.696535
(Iteration 341 / 4900) loss: 1.771244
(Iteration 351 / 4900) loss: 1.797578
(Iteration 361 / 4900) loss: 1.801109
(Iteration 371 / 4900) loss: 1.607640
(Iteration 381 / 4900) loss: 1.676314
(Iteration 391 / 4900) loss: 1.624412
(Iteration 401 / 4900) loss: 1.701401
(Iteration 411 / 4900) loss: 1.775806
(Iteration 421 / 4900) loss: 1.846421
(Iteration 431 / 4900) loss: 1.789772
(Iteration 441 / 4900) loss: 1.627572
(Iteration 451 / 4900) loss: 1.614914
(Iteration 461 / 4900) loss: 1.715750
(Iteration 471 / 4900) loss: 1.659377
(Iteration 481 / 4900) loss: 1.782116
(Epoch 1 / 10) train acc: 0.408000; val_acc: 0.394000
(Iteration 491 / 4900) loss: 1.719916
(Iteration 501 / 4900) loss: 1.696771
(Iteration 511 / 4900) loss: 1.705766
(Iteration 521 / 4900) loss: 1.577591
(Iteration 531 / 4900) loss: 1.714431
(Iteration 541 / 4900) loss: 1.675987
```

```
(Iteration 551 / 4900) loss: 1.699711
(Iteration 561 / 4900) loss: 1.562030
(Iteration 571 / 4900) loss: 1.552250
(Iteration 581 / 4900) loss: 1.655670
(Iteration 591 / 4900) loss: 1.576024
(Iteration 601 / 4900) loss: 1.658693
(Iteration 611 / 4900) loss: 1.661348
(Iteration 621 / 4900) loss: 1.707443
(Iteration 631 / 4900) loss: 1.670187
(Iteration 641 / 4900) loss: 1.675327
(Iteration 651 / 4900) loss: 1.658950
(Iteration 661 / 4900) loss: 1.526738
(Iteration 671 / 4900) loss: 1.644433
(Iteration 681 / 4900) loss: 1.601374
(Iteration 691 / 4900) loss: 1.622932
(Iteration 701 / 4900) loss: 1.593502
(Iteration 711 / 4900) loss: 1.626028
(Iteration 721 / 4900) loss: 1.597813
(Iteration 731 / 4900) loss: 1.614515
(Iteration 741 / 4900) loss: 1.860257
(Iteration 751 / 4900) loss: 1.626573
(Iteration 761 / 4900) loss: 1.644052
(Iteration 771 / 4900) loss: 1.723964
(Iteration 781 / 4900) loss: 1.514406
(Iteration 791 / 4900) loss: 1.855611
(Iteration 801 / 4900) loss: 1.608952
(Iteration 811 / 4900) loss: 1.526506
(Iteration 821 / 4900) loss: 1.564900
(Iteration 831 / 4900) loss: 1.635134
(Iteration 841 / 4900) loss: 1.480707
(Iteration 851 / 4900) loss: 1.507653
(Iteration 861 / 4900) loss: 1.559588
(Iteration 871 / 4900) loss: 1.770235
(Iteration 881 / 4900) loss: 1.553688
(Iteration 891 / 4900) loss: 1.570218
(Iteration 901 / 4900) loss: 1.640055
(Iteration 911 / 4900) loss: 1.504122
(Iteration 921 / 4900) loss: 1.475103
(Iteration 931 / 4900) loss: 1.655451
(Iteration 941 / 4900) loss: 1.540384
(Iteration 951 / 4900) loss: 1.872676
(Iteration 961 / 4900) loss: 1.650782
(Iteration 971 / 4900) loss: 1.422746
(Epoch 2 / 10) train acc: 0.433000; val_acc: 0.424000
(Iteration 981 / 4900) loss: 1.412595
(Iteration 991 / 4900) loss: 1.506664
(Iteration 1001 / 4900) loss: 1.580426
(Iteration 1011 / 4900) loss: 1.656954
(Iteration 1021 / 4900) loss: 1.540477
(Iteration 1031 / 4900) loss: 1.726451
(Iteration 1041 / 4900) loss: 1.572197
(Iteration 1051 / 4900) loss: 1.514043
(Iteration 1061 / 4900) loss: 1.600452
(Iteration 1071 / 4900) loss: 1.591877
(Iteration 1081 / 4900) loss: 1.509088
(Iteration 1091 / 4900) loss: 1.539264
(Iteration 1101 / 4900) loss: 1.567996
```

```
(Iteration 1111 / 4900) loss: 1.484264
(Iteration 1121 / 4900) loss: 1.610767
(Iteration 1131 / 4900) loss: 1.716272
(Iteration 1141 / 4900) loss: 1.482705
(Iteration 1151 / 4900) loss: 1.294015
(Iteration 1161 / 4900) loss: 1.377118
(Iteration 1171 / 4900) loss: 1.571038
(Iteration 1181 / 4900) loss: 1.572750
(Iteration 1191 / 4900) loss: 1.557778
(Iteration 1201 / 4900) loss: 1.506682
(Iteration 1211 / 4900) loss: 1.518128
(Iteration 1221 / 4900) loss: 1.851745
(Iteration 1231 / 4900) loss: 1.573468
(Iteration 1241 / 4900) loss: 1.554497
(Iteration 1251 / 4900) loss: 1.574093
(Iteration 1261 / 4900) loss: 1.468259
(Iteration 1271 / 4900) loss: 1.631064
(Iteration 1281 / 4900) loss: 1.581617
(Iteration 1291 / 4900) loss: 1.341266
(Iteration 1301 / 4900) loss: 1.599526
(Iteration 1311 / 4900) loss: 1.681465
(Iteration 1321 / 4900) loss: 1.529550
(Iteration 1331 / 4900) loss: 1.457270
(Iteration 1341 / 4900) loss: 1.398135
(Iteration 1351 / 4900) loss: 1.732383
(Iteration 1361 / 4900) loss: 1.524504
(Iteration 1371 / 4900) loss: 1.680234
(Iteration 1381 / 4900) loss: 1.613531
(Iteration 1391 / 4900) loss: 1.514638
(Iteration 1401 / 4900) loss: 1.365035
(Iteration 1411 / 4900) loss: 1.420527
(Iteration 1421 / 4900) loss: 1.502329
(Iteration 1431 / 4900) loss: 1.601997
(Iteration 1441 / 4900) loss: 1.609411
(Iteration 1451 / 4900) loss: 1.390723
(Iteration 1461 / 4900) loss: 1.413456
(Epoch 3 / 10) train acc: 0.415000; val_acc: 0.431000
(Iteration 1471 / 4900) loss: 1.390317
(Iteration 1481 / 4900) loss: 1.749006
(Iteration 1491 / 4900) loss: 1.308604
(Iteration 1501 / 4900) loss: 1.566262
(Iteration 1511 / 4900) loss: 1.822948
(Iteration 1521 / 4900) loss: 1.629824
(Iteration 1531 / 4900) loss: 1.467139
(Iteration 1541 / 4900) loss: 1.511465
(Iteration 1551 / 4900) loss: 1.574243
(Iteration 1561 / 4900) loss: 1.456230
(Iteration 1571 / 4900) loss: 1.442357
(Iteration 1581 / 4900) loss: 1.559810
(Iteration 1591 / 4900) loss: 1.658602
(Iteration 1601 / 4900) loss: 1.511004
(Iteration 1611 / 4900) loss: 1.462563
(Iteration 1621 / 4900) loss: 1.470672
(Iteration 1631 / 4900) loss: 1.517981
(Iteration 1641 / 4900) loss: 1.623759
(Iteration 1651 / 4900) loss: 1.400474
(Iteration 1661 / 4900) loss: 1.432484
```

```
(Iteration 1671 / 4900) loss: 1.520222
(Iteration 1681 / 4900) loss: 1.518606
(Iteration 1691 / 4900) loss: 1.481267
(Iteration 1701 / 4900) loss: 1.521433
(Iteration 1711 / 4900) loss: 1.469645
(Iteration 1721 / 4900) loss: 1.609355
(Iteration 1731 / 4900) loss: 1.413776
(Iteration 1741 / 4900) loss: 1.579957
(Iteration 1751 / 4900) loss: 1.455680
(Iteration 1761 / 4900) loss: 1.399787
(Iteration 1771 / 4900) loss: 1.361812
(Iteration 1781 / 4900) loss: 1.733088
(Iteration 1791 / 4900) loss: 1.609184
(Iteration 1801 / 4900) loss: 1.678917
(Iteration 1811 / 4900) loss: 1.580328
(Iteration 1821 / 4900) loss: 1.392046
(Iteration 1831 / 4900) loss: 1.486765
(Iteration 1841 / 4900) loss: 1.709290
(Iteration 1851 / 4900) loss: 1.399190
(Iteration 1861 / 4900) loss: 1.488129
(Iteration 1871 / 4900) loss: 1.509695
(Iteration 1881 / 4900) loss: 1.456331
(Iteration 1891 / 4900) loss: 1.703417
(Iteration 1901 / 4900) loss: 1.557781
(Iteration 1911 / 4900) loss: 1.667922
(Iteration 1921 / 4900) loss: 1.420304
(Iteration 1931 / 4900) loss: 1.398344
(Iteration 1941 / 4900) loss: 1.394595
(Iteration 1951 / 4900) loss: 1.503018
(Epoch 4 / 10) train acc: 0.454000; val_acc: 0.414000
(Iteration 1961 / 4900) loss: 1.658352
(Iteration 1971 / 4900) loss: 1.380106
(Iteration 1981 / 4900) loss: 1.314798
(Iteration 1991 / 4900) loss: 1.352132
(Iteration 2001 / 4900) loss: 1.378944
(Iteration 2011 / 4900) loss: 1.519382
(Iteration 2021 / 4900) loss: 1.440203
(Iteration 2031 / 4900) loss: 1.520978
(Iteration 2041 / 4900) loss: 1.409114
(Iteration 2051 / 4900) loss: 1.530329
(Iteration 2061 / 4900) loss: 1.393353
(Iteration 2071 / 4900) loss: 1.452324
(Iteration 2081 / 4900) loss: 1.376602
(Iteration 2091 / 4900) loss: 1.629977
(Iteration 2101 / 4900) loss: 1.520669
(Iteration 2111 / 4900) loss: 1.671368
(Iteration 2121 / 4900) loss: 1.281485
(Iteration 2131 / 4900) loss: 1.479481
(Iteration 2141 / 4900) loss: 1.296559
(Iteration 2151 / 4900) loss: 1.217426
(Iteration 2161 / 4900) loss: 1.338424
(Iteration 2171 / 4900) loss: 1.354330
(Iteration 2181 / 4900) loss: 1.608985
(Iteration 2191 / 4900) loss: 1.531467
(Iteration 2201 / 4900) loss: 1.352534
(Iteration 2211 / 4900) loss: 1.509669
(Iteration 2221 / 4900) loss: 1.418174
```

```
(Iteration 2231 / 4900) loss: 1.547347
(Iteration 2241 / 4900) loss: 1.660057
(Iteration 2251 / 4900) loss: 1.439596
(Iteration 2261 / 4900) loss: 1.570417
(Iteration 2271 / 4900) loss: 1.323973
(Iteration 2281 / 4900) loss: 1.763992
(Iteration 2291 / 4900) loss: 1.502648
(Iteration 2301 / 4900) loss: 1.401639
(Iteration 2311 / 4900) loss: 1.552625
(Iteration 2321 / 4900) loss: 1.607425
(Iteration 2331 / 4900) loss: 1.363481
(Iteration 2341 / 4900) loss: 1.447289
(Iteration 2351 / 4900) loss: 1.605258
(Iteration 2361 / 4900) loss: 1.456590
(Iteration 2371 / 4900) loss: 1.663337
(Iteration 2381 / 4900) loss: 1.466471
(Iteration 2391 / 4900) loss: 1.626188
(Iteration 2401 / 4900) loss: 1.417077
(Iteration 2411 / 4900) loss: 1.347546
(Iteration 2421 / 4900) loss: 1.472374
(Iteration 2431 / 4900) loss: 1.654754
(Iteration 2441 / 4900) loss: 1.699059
(Epoch 5 / 10) train acc: 0.468000; val_acc: 0.460000
(Iteration 2451 / 4900) loss: 1.570660
(Iteration 2461 / 4900) loss: 1.651585
(Iteration 2471 / 4900) loss: 1.358496
(Iteration 2481 / 4900) loss: 1.339270
(Iteration 2491 / 4900) loss: 1.511960
(Iteration 2501 / 4900) loss: 1.662425
(Iteration 2511 / 4900) loss: 1.387705
(Iteration 2521 / 4900) loss: 1.420369
(Iteration 2531 / 4900) loss: 1.326955
(Iteration 2541 / 4900) loss: 1.367641
(Iteration 2551 / 4900) loss: 1.403026
(Iteration 2561 / 4900) loss: 1.792609
(Iteration 2571 / 4900) loss: 1.333480
(Iteration 2581 / 4900) loss: 1.347500
(Iteration 2591 / 4900) loss: 1.524510
(Iteration 2601 / 4900) loss: 1.442210
(Iteration 2611 / 4900) loss: 1.483822
(Iteration 2621 / 4900) loss: 1.526979
(Iteration 2631 / 4900) loss: 1.518453
(Iteration 2641 / 4900) loss: 1.796644
(Iteration 2651 / 4900) loss: 1.337431
(Iteration 2661 / 4900) loss: 1.401175
(Iteration 2671 / 4900) loss: 1.396547
(Iteration 2681 / 4900) loss: 1.614981
(Iteration 2691 / 4900) loss: 1.416721
(Iteration 2701 / 4900) loss: 1.333179
(Iteration 2711 / 4900) loss: 1.515608
(Iteration 2721 / 4900) loss: 1.532201
(Iteration 2731 / 4900) loss: 1.326058
(Iteration 2741 / 4900) loss: 1.298951
(Iteration 2751 / 4900) loss: 1.519663
(Iteration 2761 / 4900) loss: 1.414386
(Iteration 2771 / 4900) loss: 1.390696
(Iteration 2781 / 4900) loss: 1.527466
```

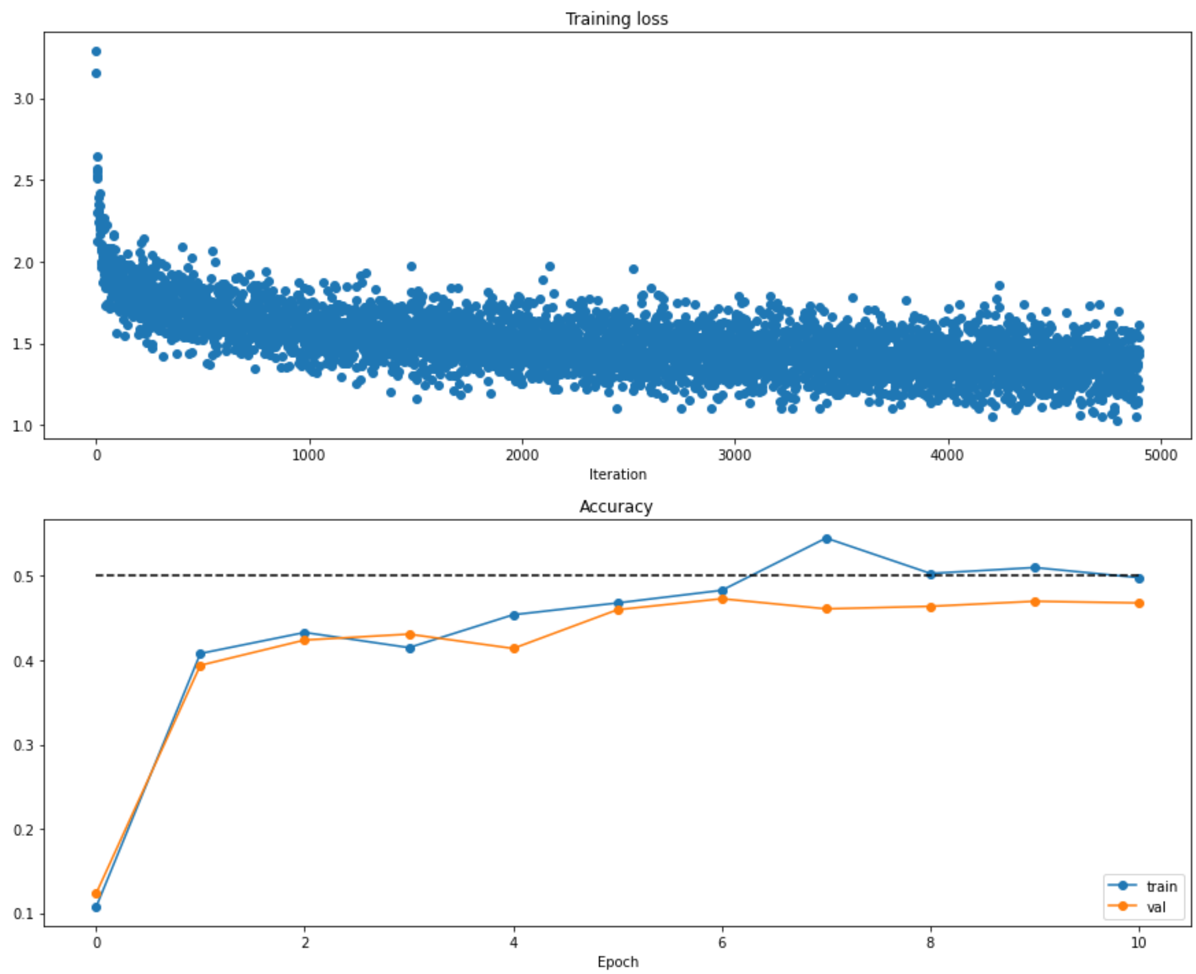
```
(Iteration 2791 / 4900) loss: 1.472511
(Iteration 2801 / 4900) loss: 1.568581
(Iteration 2811 / 4900) loss: 1.354628
(Iteration 2821 / 4900) loss: 1.525698
(Iteration 2831 / 4900) loss: 1.505145
(Iteration 2841 / 4900) loss: 1.474250
(Iteration 2851 / 4900) loss: 1.467496
(Iteration 2861 / 4900) loss: 1.358249
(Iteration 2871 / 4900) loss: 1.266399
(Iteration 2881 / 4900) loss: 1.443252
(Iteration 2891 / 4900) loss: 1.360145
(Iteration 2901 / 4900) loss: 1.244391
(Iteration 2911 / 4900) loss: 1.375927
(Iteration 2921 / 4900) loss: 1.281383
(Iteration 2931 / 4900) loss: 1.503631
(Epoch 6 / 10) train acc: 0.483000; val_acc: 0.473000
(Iteration 2941 / 4900) loss: 1.508058
(Iteration 2951 / 4900) loss: 1.627976
(Iteration 2961 / 4900) loss: 1.417924
(Iteration 2971 / 4900) loss: 1.290509
(Iteration 2981 / 4900) loss: 1.549190
(Iteration 2991 / 4900) loss: 1.368109
(Iteration 3001 / 4900) loss: 1.336898
(Iteration 3011 / 4900) loss: 1.274182
(Iteration 3021 / 4900) loss: 1.187609
(Iteration 3031 / 4900) loss: 1.589349
(Iteration 3041 / 4900) loss: 1.463505
(Iteration 3051 / 4900) loss: 1.479151
(Iteration 3061 / 4900) loss: 1.504976
(Iteration 3071 / 4900) loss: 1.614850
(Iteration 3081 / 4900) loss: 1.268689
(Iteration 3091 / 4900) loss: 1.249049
(Iteration 3101 / 4900) loss: 1.324033
(Iteration 3111 / 4900) loss: 1.436996
(Iteration 3121 / 4900) loss: 1.424637
(Iteration 3131 / 4900) loss: 1.412341
(Iteration 3141 / 4900) loss: 1.278621
(Iteration 3151 / 4900) loss: 1.475220
(Iteration 3161 / 4900) loss: 1.377523
(Iteration 3171 / 4900) loss: 1.406185
(Iteration 3181 / 4900) loss: 1.601938
(Iteration 3191 / 4900) loss: 1.564705
(Iteration 3201 / 4900) loss: 1.262510
(Iteration 3211 / 4900) loss: 1.437562
(Iteration 3221 / 4900) loss: 1.213535
(Iteration 3231 / 4900) loss: 1.323498
(Iteration 3241 / 4900) loss: 1.254896
(Iteration 3251 / 4900) loss: 1.559102
(Iteration 3261 / 4900) loss: 1.532790
(Iteration 3271 / 4900) loss: 1.101451
(Iteration 3281 / 4900) loss: 1.442550
(Iteration 3291 / 4900) loss: 1.336728
(Iteration 3301 / 4900) loss: 1.479650
(Iteration 3311 / 4900) loss: 1.590911
(Iteration 3321 / 4900) loss: 1.489436
(Iteration 3331 / 4900) loss: 1.686835
(Iteration 3341 / 4900) loss: 1.434351
```

```
(Iteration 3351 / 4900) loss: 1.402058
(Iteration 3361 / 4900) loss: 1.693258
(Iteration 3371 / 4900) loss: 1.524842
(Iteration 3381 / 4900) loss: 1.537934
(Iteration 3391 / 4900) loss: 1.220046
(Iteration 3401 / 4900) loss: 1.107242
(Iteration 3411 / 4900) loss: 1.285582
(Iteration 3421 / 4900) loss: 1.518551
(Epoch 7 / 10) train acc: 0.545000; val_acc: 0.461000
(Iteration 3431 / 4900) loss: 1.451782
(Iteration 3441 / 4900) loss: 1.578873
(Iteration 3451 / 4900) loss: 1.483952
(Iteration 3461 / 4900) loss: 1.517126
(Iteration 3471 / 4900) loss: 1.361448
(Iteration 3481 / 4900) loss: 1.392112
(Iteration 3491 / 4900) loss: 1.404703
(Iteration 3501 / 4900) loss: 1.371145
(Iteration 3511 / 4900) loss: 1.417723
(Iteration 3521 / 4900) loss: 1.503432
(Iteration 3531 / 4900) loss: 1.327544
(Iteration 3541 / 4900) loss: 1.542255
(Iteration 3551 / 4900) loss: 1.289399
(Iteration 3561 / 4900) loss: 1.311728
(Iteration 3571 / 4900) loss: 1.385101
(Iteration 3581 / 4900) loss: 1.295868
(Iteration 3591 / 4900) loss: 1.502512
(Iteration 3601 / 4900) loss: 1.297429
(Iteration 3611 / 4900) loss: 1.440759
(Iteration 3621 / 4900) loss: 1.405377
(Iteration 3631 / 4900) loss: 1.604754
(Iteration 3641 / 4900) loss: 1.529330
(Iteration 3651 / 4900) loss: 1.365253
(Iteration 3661 / 4900) loss: 1.547297
(Iteration 3671 / 4900) loss: 1.357217
(Iteration 3681 / 4900) loss: 1.308929
(Iteration 3691 / 4900) loss: 1.637895
(Iteration 3701 / 4900) loss: 1.386182
(Iteration 3711 / 4900) loss: 1.378171
(Iteration 3721 / 4900) loss: 1.575915
(Iteration 3731 / 4900) loss: 1.612256
(Iteration 3741 / 4900) loss: 1.104872
(Iteration 3751 / 4900) loss: 1.260065
(Iteration 3761 / 4900) loss: 1.432096
(Iteration 3771 / 4900) loss: 1.492711
(Iteration 3781 / 4900) loss: 1.180341
(Iteration 3791 / 4900) loss: 1.396191
(Iteration 3801 / 4900) loss: 1.237356
(Iteration 3811 / 4900) loss: 1.291746
(Iteration 3821 / 4900) loss: 1.297762
(Iteration 3831 / 4900) loss: 1.462655
(Iteration 3841 / 4900) loss: 1.475105
(Iteration 3851 / 4900) loss: 1.315840
(Iteration 3861 / 4900) loss: 1.424814
(Iteration 3871 / 4900) loss: 1.340077
(Iteration 3881 / 4900) loss: 1.191771
(Iteration 3891 / 4900) loss: 1.523963
(Iteration 3901 / 4900) loss: 1.339174
```

```
(Iteration 3911 / 4900) loss: 1.376735
(Epoch 8 / 10) train acc: 0.503000; val_acc: 0.464000
(Iteration 3921 / 4900) loss: 1.610358
(Iteration 3931 / 4900) loss: 1.246343
(Iteration 3941 / 4900) loss: 1.240188
(Iteration 3951 / 4900) loss: 1.362902
(Iteration 3961 / 4900) loss: 1.248327
(Iteration 3971 / 4900) loss: 1.308736
(Iteration 3981 / 4900) loss: 1.185860
(Iteration 3991 / 4900) loss: 1.446666
(Iteration 4001 / 4900) loss: 1.471682
(Iteration 4011 / 4900) loss: 1.369916
(Iteration 4021 / 4900) loss: 1.370456
(Iteration 4031 / 4900) loss: 1.430832
(Iteration 4041 / 4900) loss: 1.313023
(Iteration 4051 / 4900) loss: 1.390870
(Iteration 4061 / 4900) loss: 1.309779
(Iteration 4071 / 4900) loss: 1.403723
(Iteration 4081 / 4900) loss: 1.338327
(Iteration 4091 / 4900) loss: 1.480707
(Iteration 4101 / 4900) loss: 1.429648
(Iteration 4111 / 4900) loss: 1.317005
(Iteration 4121 / 4900) loss: 1.440012
(Iteration 4131 / 4900) loss: 1.288775
(Iteration 4141 / 4900) loss: 1.560772
(Iteration 4151 / 4900) loss: 1.354293
(Iteration 4161 / 4900) loss: 1.440708
(Iteration 4171 / 4900) loss: 1.591189
(Iteration 4181 / 4900) loss: 1.275631
(Iteration 4191 / 4900) loss: 1.446094
(Iteration 4201 / 4900) loss: 1.548837
(Iteration 4211 / 4900) loss: 1.348864
(Iteration 4221 / 4900) loss: 1.352441
(Iteration 4231 / 4900) loss: 1.770951
(Iteration 4241 / 4900) loss: 1.721834
(Iteration 4251 / 4900) loss: 1.299906
(Iteration 4261 / 4900) loss: 1.285114
(Iteration 4271 / 4900) loss: 1.158092
(Iteration 4281 / 4900) loss: 1.523972
(Iteration 4291 / 4900) loss: 1.239014
(Iteration 4301 / 4900) loss: 1.360439
(Iteration 4311 / 4900) loss: 1.445667
(Iteration 4321 / 4900) loss: 1.529782
(Iteration 4331 / 4900) loss: 1.345141
(Iteration 4341 / 4900) loss: 1.373862
(Iteration 4351 / 4900) loss: 1.466385
(Iteration 4361 / 4900) loss: 1.350607
(Iteration 4371 / 4900) loss: 1.378573
(Iteration 4381 / 4900) loss: 1.252644
(Iteration 4391 / 4900) loss: 1.662401
(Iteration 4401 / 4900) loss: 1.342831
(Epoch 9 / 10) train acc: 0.510000; val_acc: 0.470000
(Iteration 4411 / 4900) loss: 1.343880
(Iteration 4421 / 4900) loss: 1.457136
(Iteration 4431 / 4900) loss: 1.432976
(Iteration 4441 / 4900) loss: 1.187305
(Iteration 4451 / 4900) loss: 1.220427
```

```
(Iteration 4461 / 4900) loss: 1.698173
(Iteration 4471 / 4900) loss: 1.369832
(Iteration 4481 / 4900) loss: 1.506724
(Iteration 4491 / 4900) loss: 1.417483
(Iteration 4501 / 4900) loss: 1.412963
(Iteration 4511 / 4900) loss: 1.542774
(Iteration 4521 / 4900) loss: 1.411619
(Iteration 4531 / 4900) loss: 1.178885
(Iteration 4541 / 4900) loss: 1.197942
(Iteration 4551 / 4900) loss: 1.349370
(Iteration 4561 / 4900) loss: 1.334170
(Iteration 4571 / 4900) loss: 1.501928
(Iteration 4581 / 4900) loss: 1.402278
(Iteration 4591 / 4900) loss: 1.366862
(Iteration 4601 / 4900) loss: 1.472537
(Iteration 4611 / 4900) loss: 1.353501
(Iteration 4621 / 4900) loss: 1.426901
(Iteration 4631 / 4900) loss: 1.413638
(Iteration 4641 / 4900) loss: 1.338972
(Iteration 4651 / 4900) loss: 1.520199
(Iteration 4661 / 4900) loss: 1.520844
(Iteration 4671 / 4900) loss: 1.363972
(Iteration 4681 / 4900) loss: 1.316547
(Iteration 4691 / 4900) loss: 1.213473
(Iteration 4701 / 4900) loss: 1.386295
(Iteration 4711 / 4900) loss: 1.387165
(Iteration 4721 / 4900) loss: 1.191210
(Iteration 4731 / 4900) loss: 1.288106
(Iteration 4741 / 4900) loss: 1.367852
(Iteration 4751 / 4900) loss: 1.305922
(Iteration 4761 / 4900) loss: 1.395860
(Iteration 4771 / 4900) loss: 1.068991
(Iteration 4781 / 4900) loss: 1.394952
(Iteration 4791 / 4900) loss: 1.381461
(Iteration 4801 / 4900) loss: 1.440518
(Iteration 4811 / 4900) loss: 1.434127
(Iteration 4821 / 4900) loss: 1.515883
(Iteration 4831 / 4900) loss: 1.447357
(Iteration 4841 / 4900) loss: 1.236491
(Iteration 4851 / 4900) loss: 1.159709
(Iteration 4861 / 4900) loss: 1.408492
(Iteration 4871 / 4900) loss: 1.394239
(Iteration 4881 / 4900) loss: 1.283875
(Iteration 4891 / 4900) loss: 1.132839
(Epoch 10 / 10) train acc: 0.498000; val_acc: 0.468000
```

In [13]:
```
# 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 [14]:
```
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(grad_num,grads[name]*X.shape[0])
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[name
])))
```

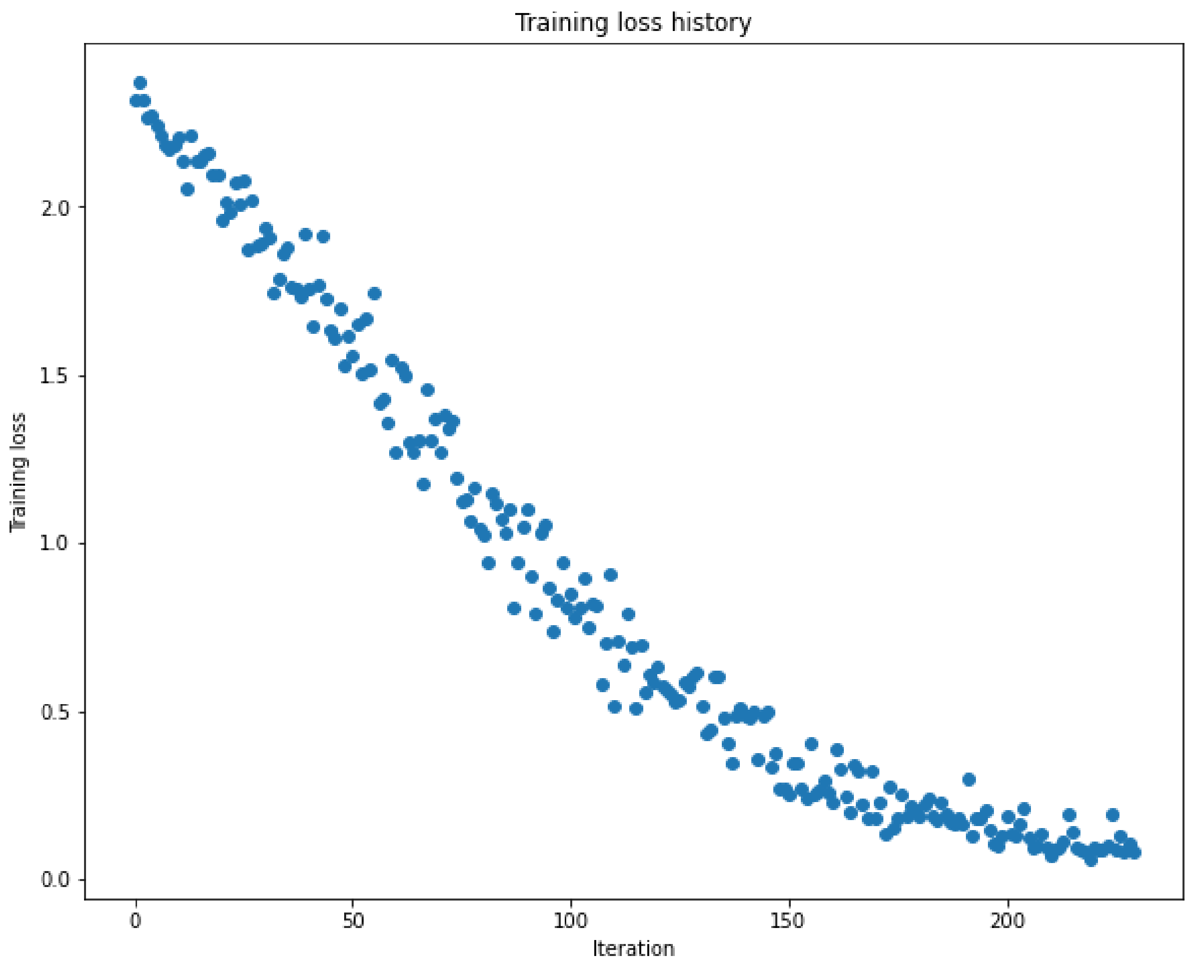
```
Running check with reg = 0
Initial loss: 2.2968331912697075
W0 relative error: 3.6326444228315726e-08
W1 relative error: 1.5131639813011978e-06
W2 relative error: 1.2143928868677336e-06
b0 relative error: 3.3810222914195847e-09
b1 relative error: 2.6252880018410976e-09
b2 relative error: 1.502397065292582e-10
Running check with reg = 3.14
Initial loss: 7.140821385251918
W0 relative error: 2.9472455013329315e-07
W1 relative error: 1.0760668517647348e-07
W2 relative error: 2.1939681139615717e-08
b0 relative error: 1.791998126910202e-08
b1 relative error: 5.020177751466017e-08
b2 relative error: 1.3193392969855164e-10
```

In [16]:
```
# 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-3

model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)

solver = Solver(model, small_data,
                print_every=10, num_epochs=115, 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 / 230) loss: 2.319922
(Epoch 0 / 115) train acc: 0.080000; val_acc: 0.081000
(Epoch 1 / 115) train acc: 0.080000; val_acc: 0.086000
(Epoch 2 / 115) train acc: 0.260000; val_acc: 0.094000
(Epoch 3 / 115) train acc: 0.320000; val_acc: 0.111000
(Epoch 4 / 115) train acc: 0.300000; val_acc: 0.109000
(Epoch 5 / 115) train acc: 0.320000; val_acc: 0.116000
(Iteration 11 / 230) loss: 2.206508
(Epoch 6 / 115) train acc: 0.340000; val_acc: 0.124000
(Epoch 7 / 115) train acc: 0.340000; val_acc: 0.125000
(Epoch 8 / 115) train acc: 0.400000; val_acc: 0.128000
(Epoch 9 / 115) train acc: 0.380000; val_acc: 0.129000
(Epoch 10 / 115) train acc: 0.460000; val_acc: 0.126000
(Iteration 21 / 230) loss: 1.961897
(Epoch 11 / 115) train acc: 0.480000; val_acc: 0.121000
(Epoch 12 / 115) train acc: 0.500000; val_acc: 0.127000
(Epoch 13 / 115) train acc: 0.540000; val_acc: 0.132000
(Epoch 14 / 115) train acc: 0.540000; val_acc: 0.138000
(Epoch 15 / 115) train acc: 0.480000; val_acc: 0.139000
(Iteration 31 / 230) loss: 1.940253
(Epoch 16 / 115) train acc: 0.540000; val_acc: 0.138000
(Epoch 17 / 115) train acc: 0.540000; val_acc: 0.144000
(Epoch 18 / 115) train acc: 0.560000; val_acc: 0.145000
(Epoch 19 / 115) train acc: 0.540000; val_acc: 0.137000
(Epoch 20 / 115) train acc: 0.560000; val_acc: 0.141000
(Iteration 41 / 230) loss: 1.755426
(Epoch 21 / 115) train acc: 0.580000; val_acc: 0.142000
(Epoch 22 / 115) train acc: 0.620000; val_acc: 0.145000
(Epoch 23 / 115) train acc: 0.620000; val_acc: 0.143000
(Epoch 24 / 115) train acc: 0.600000; val_acc: 0.148000
(Epoch 25 / 115) train acc: 0.600000; val_acc: 0.146000
(Iteration 51 / 230) loss: 1.558535
(Epoch 26 / 115) train acc: 0.600000; val_acc: 0.144000
(Epoch 27 / 115) train acc: 0.600000; val_acc: 0.146000
(Epoch 28 / 115) train acc: 0.660000; val_acc: 0.145000
(Epoch 29 / 115) train acc: 0.640000; val_acc: 0.141000
(Epoch 30 / 115) train acc: 0.640000; val_acc: 0.147000
(Iteration 61 / 230) loss: 1.269664
(Epoch 31 / 115) train acc: 0.620000; val_acc: 0.151000
(Epoch 32 / 115) train acc: 0.680000; val_acc: 0.157000
(Epoch 33 / 115) train acc: 0.660000; val_acc: 0.160000
(Epoch 34 / 115) train acc: 0.700000; val_acc: 0.163000
(Epoch 35 / 115) train acc: 0.760000; val_acc: 0.168000
(Iteration 71 / 230) loss: 1.270540
(Epoch 36 / 115) train acc: 0.780000; val_acc: 0.174000
(Epoch 37 / 115) train acc: 0.780000; val_acc: 0.160000
(Epoch 38 / 115) train acc: 0.800000; val_acc: 0.175000
(Epoch 39 / 115) train acc: 0.800000; val_acc: 0.164000
(Epoch 40 / 115) train acc: 0.800000; val_acc: 0.162000
(Iteration 81 / 230) loss: 1.026558
(Epoch 41 / 115) train acc: 0.800000; val_acc: 0.158000
(Epoch 42 / 115) train acc: 0.820000; val_acc: 0.156000
(Epoch 43 / 115) train acc: 0.820000; val_acc: 0.155000
(Epoch 44 / 115) train acc: 0.840000; val_acc: 0.155000
(Epoch 45 / 115) train acc: 0.840000; val_acc: 0.151000
(Iteration 91 / 230) loss: 1.101510
(Epoch 46 / 115) train acc: 0.860000; val_acc: 0.152000
```

```
(Epoch 47 / 115) train acc: 0.900000; val_acc: 0.156000
(Epoch 48 / 115) train acc: 0.880000; val_acc: 0.160000
(Epoch 49 / 115) train acc: 0.900000; val_acc: 0.163000
(Epoch 50 / 115) train acc: 0.900000; val_acc: 0.165000
(Iteration 101 / 230) loss: 0.846601
(Epoch 51 / 115) train acc: 0.900000; val_acc: 0.166000
(Epoch 52 / 115) train acc: 0.920000; val_acc: 0.163000
(Epoch 53 / 115) train acc: 0.900000; val_acc: 0.162000
(Epoch 54 / 115) train acc: 0.920000; val_acc: 0.160000
(Epoch 55 / 115) train acc: 0.940000; val_acc: 0.162000
(Iteration 111 / 230) loss: 0.517078
(Epoch 56 / 115) train acc: 0.920000; val_acc: 0.169000
(Epoch 57 / 115) train acc: 0.960000; val_acc: 0.169000
(Epoch 58 / 115) train acc: 0.960000; val_acc: 0.169000
(Epoch 59 / 115) train acc: 0.960000; val_acc: 0.166000
(Epoch 60 / 115) train acc: 0.960000; val_acc: 0.174000
(Iteration 121 / 230) loss: 0.630985
(Epoch 61 / 115) train acc: 0.960000; val_acc: 0.166000
(Epoch 62 / 115) train acc: 0.960000; val_acc: 0.166000
(Epoch 63 / 115) train acc: 0.960000; val_acc: 0.168000
(Epoch 64 / 115) train acc: 0.960000; val_acc: 0.172000
(Epoch 65 / 115) train acc: 0.960000; val_acc: 0.169000
(Iteration 131 / 230) loss: 0.513243
(Epoch 66 / 115) train acc: 0.980000; val_acc: 0.174000
(Epoch 67 / 115) train acc: 0.960000; val_acc: 0.180000
(Epoch 68 / 115) train acc: 0.960000; val_acc: 0.171000
(Epoch 69 / 115) train acc: 0.960000; val_acc: 0.177000
(Epoch 70 / 115) train acc: 1.000000; val_acc: 0.178000
(Iteration 141 / 230) loss: 0.486122
(Epoch 71 / 115) train acc: 1.000000; val_acc: 0.178000
(Epoch 72 / 115) train acc: 1.000000; val_acc: 0.178000
(Epoch 73 / 115) train acc: 1.000000; val_acc: 0.172000
(Epoch 74 / 115) train acc: 1.000000; val_acc: 0.171000
(Epoch 75 / 115) train acc: 1.000000; val_acc: 0.173000
(Iteration 151 / 230) loss: 0.253726
(Epoch 76 / 115) train acc: 1.000000; val_acc: 0.172000
(Epoch 77 / 115) train acc: 1.000000; val_acc: 0.180000
(Epoch 78 / 115) train acc: 1.000000; val_acc: 0.175000
(Epoch 79 / 115) train acc: 1.000000; val_acc: 0.181000
(Epoch 80 / 115) train acc: 1.000000; val_acc: 0.179000
(Iteration 161 / 230) loss: 0.230596
(Epoch 81 / 115) train acc: 1.000000; val_acc: 0.181000
(Epoch 82 / 115) train acc: 1.000000; val_acc: 0.177000
(Epoch 83 / 115) train acc: 1.000000; val_acc: 0.180000
(Epoch 84 / 115) train acc: 1.000000; val_acc: 0.173000
(Epoch 85 / 115) train acc: 1.000000; val_acc: 0.181000
(Iteration 171 / 230) loss: 0.181252
(Epoch 86 / 115) train acc: 1.000000; val_acc: 0.183000
(Epoch 87 / 115) train acc: 1.000000; val_acc: 0.179000
(Epoch 88 / 115) train acc: 1.000000; val_acc: 0.178000
(Epoch 89 / 115) train acc: 1.000000; val_acc: 0.180000
(Epoch 90 / 115) train acc: 1.000000; val_acc: 0.177000
(Iteration 181 / 230) loss: 0.185034
(Epoch 91 / 115) train acc: 1.000000; val_acc: 0.174000
(Epoch 92 / 115) train acc: 1.000000; val_acc: 0.175000
(Epoch 93 / 115) train acc: 1.000000; val_acc: 0.173000
(Epoch 94 / 115) train acc: 1.000000; val_acc: 0.181000
```

```
(Epoch 95 / 115) train acc: 1.000000; val_acc: 0.180000
(Iteration 191 / 230) loss: 0.164014
(Epoch 96 / 115) train acc: 1.000000; val_acc: 0.180000
(Epoch 97 / 115) train acc: 1.000000; val_acc: 0.179000
(Epoch 98 / 115) train acc: 1.000000; val_acc: 0.186000
(Epoch 99 / 115) train acc: 1.000000; val_acc: 0.182000
(Epoch 100 / 115) train acc: 1.000000; val_acc: 0.181000
(Iteration 201 / 230) loss: 0.185446
(Epoch 101 / 115) train acc: 1.000000; val_acc: 0.179000
(Epoch 102 / 115) train acc: 1.000000; val_acc: 0.182000
(Epoch 103 / 115) train acc: 1.000000; val_acc: 0.178000
(Epoch 104 / 115) train acc: 1.000000; val_acc: 0.183000
(Epoch 105 / 115) train acc: 1.000000; val_acc: 0.185000
(Iteration 211 / 230) loss: 0.067611
(Epoch 106 / 115) train acc: 1.000000; val_acc: 0.183000
(Epoch 107 / 115) train acc: 1.000000; val_acc: 0.179000
(Epoch 108 / 115) train acc: 1.000000; val_acc: 0.181000
(Epoch 109 / 115) train acc: 1.000000; val_acc: 0.180000
(Epoch 110 / 115) train acc: 1.000000; val_acc: 0.180000
(Iteration 221 / 230) loss: 0.093625
(Epoch 111 / 115) train acc: 1.000000; val_acc: 0.185000
(Epoch 112 / 115) train acc: 1.000000; val_acc: 0.185000
(Epoch 113 / 115) train acc: 1.000000; val_acc: 0.183000
(Epoch 114 / 115) train acc: 1.000000; val_acc: 0.181000
(Epoch 115 / 115) train acc: 1.000000; val_acc: 0.185000
```



```python
import numpy as np
import pdb

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


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

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

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

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

  # ================================================================ #
  # 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]
  out = np.matmul(x.reshape(N,-1),w)+b
  pass

  # ================================================================ #
  # 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)
```

```python
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)

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

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

    # dout is N x M
    # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after resh
    # db should be M; it is just the sum over dout examples
    N = x.shape[0]
    x = x.reshape(N,-1)

    M = b.shape[0]
    db = np.sum(dout,0)
    dx = np.matmul(dout,w.T)
    dw = np.matmul(x.T,dout)
    pass

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

    return dx, dw, db

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

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ================================================================ #
    out = np.clip(x,a_min=0,a_max=float('inf'))
    pass
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

```
    cache = x
    return out, cache


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

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

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

    # =============================================================== #
    # YOUR CODE HERE:
    #   Implement the ReLU backward pass
    # =============================================================== #
    # ReLU directs linearly to those > 0
    dx = np.array(x>=0,dtype=np.float)*dout
    pass

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

    return dx

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

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

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

```python
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
```

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

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

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

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

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

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

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

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

```python
    self.params = {}
    self.params['W1'] = weight_scale * np.random.randn(input_dim,hidden_dims)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = weight_scale * np.random.randn(hidden_dims,num_classes)
    self.params['b2'] = np.zeros(num_classes)

    pass

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

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

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

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

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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ================================================================ #
    N = X.shape[0]
    affine1,affine1_cache = affine_forward(X,self.params['W1'],self.params['b1'])
    relu1,relu1_cache = relu_forward(affine1)

    affine2,affine2_cache = affine_forward(relu1,self.params['W2'],self.params['b2'])
    scores = affine2
    pass
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

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

    loss, grads = 0, {}
    # ================================================================ #
    # YOUR CODE HERE:
```

```
    #   Implement the backward pass of the two-layer neural net.  Store
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ================================================================ #
    Z = np.exp(scores)/np.sum(np.exp(scores),1)[:,np.newaxis]
    dLdz = np.copy(Z)
    dLdz[np.arange(N),y] = dLdz[np.arange(N),y] - 1
    dLdz = dLdz*1/N


    dx2,dw2,db2 = affine_backward(dLdz,affine2_cache)
    relu_grad1 = relu_backward(dx2,relu1_cache)
    dx1,dw1,db1 = affine_backward(relu_grad1,affine1_cache)



    reg_loss = 0.5 * (np.linalg.norm(self.params['W1'])**2 + np.linalg.norm(self.params['W2'])**2)
    softmax_loss = np.mean(-np.log(np.exp(scores[np.arange(N),y])/np.sum(np.exp(scores),1)))
    loss = softmax_loss + self.reg*reg_loss
    #affine_backward(affine2_cache)

    grads['W1'] = dw1 + 0.5*self.reg*2*self.params['W1']
    grads['W2'] = dw2 + 0.5*self.reg*2*self.params['W2']
    grads['b1'] = db1
    grads['b2'] = db2


    pass

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

    return loss, grads


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

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

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

  Similar to the TwoLayerNet above, learnable parameters are stored in the
```

```
    self.params dictionary and will be learned using the Solver class.
    """

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

      Inputs:
      - hidden_dims: A list of integers giving the size of each hidden layer.
      - input_dim: An integer giving the size of the input.
      - num_classes: An integer giving the number of classes to classify.
      - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
        the network should not use dropout at all.
      - use_batchnorm: Whether or not the network should use batch normalization.
      - reg: Scalar giving L2 regularization strength.
      - weight_scale: Scalar giving the standard deviation for random
        initialization of the weights.
      - dtype: A numpy datatype object; all computations will be performed using
        this datatype. float32 is faster but less accurate, so you should use
        float64 for numeric gradient checking.
      - seed: If not None, then pass this random seed to the dropout layers. This
        will make the dropout layers deteriminstic so we can gradient check the
        model.
      """
      self.use_batchnorm = use_batchnorm
      self.use_dropout = dropout > 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 dictionary.
      #   The weights and biases of layer 1 are W1 and b1; and in general the
      #   weights and biases of layer i are Wi and bi. The
      #   biases are initialized to zero and the weights are initialized
      #   so that each parameter has mean 0 and standard deviation weight_scale.
      # ================================================================ #

      self.param_tuples = [("W{}".format(i),"b{}".format(i)) for i in np.arange(self.num_layers)]
      self.dims = [(input_dim,hidden_dims[0])]
      self.dims.extend( [(hidden_dims[i],hidden_dims[i+1]) for i in np.arange(self.num_layers-2)] )
      self.dims.append((hidden_dims[-1],num_classes))


      for i,(w,b) in enumerate(self.param_tuples):
          self.params[w] = weight_scale * np.random.randn(*self.dims[i])
          self.params[b] = np.zeros(self.dims[i][1])


      pass

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

```python
    # ================================================================ #

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

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

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


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

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

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

    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    # ================================================================ #
    N = X.shape[0]

    caches = []

    for i,(w,b) in enumerate(self.param_tuples):

        if i == (len(self.param_tuples)-1):
            X,affine_cache = affine_forward(X,self.params[w],self.params[b])
```

```
            caches.append((affine_cache,None))
            break

        X,affine_cache = affine_forward(X,self.params[w],self.params[b])
        X,relu_cache = relu_forward(X)
        caches.append((affine_cache,relu_cache))


    scores = X
    Z = np.exp(scores)/np.sum(np.exp(scores),1)[:,np.newaxis]

    pass

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

    # If test mode return early
    if mode == 'test':
      return scores

    loss, grads = 0.0, {}
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backwards pass of the FC net and store the gradients
    #   in the grads dict, so that grads[k] is the gradient of self.params[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    # ================================================================ #

    reg_loss = 0.5 * (np.sum( [np.linalg.norm(self.params[w])**2 for (w,_) in self.param_tuples]) )
    softmax_loss = np.mean(-np.log(np.exp(scores[np.arange(N),y])/np.sum(np.exp(scores),1)))
    loss = softmax_loss + self.reg*reg_loss




    dLdz = np.copy(Z)
    dLdz[np.arange(N),y] = dLdz[np.arange(N),y] - 1
    dLdz = dLdz * 1/N

    for i,(affine_cache,relu_cache) in enumerate(caches[::-1]):
        if relu_cache is None:
            dx,dw,db = affine_backward(dLdz,affine_cache)

            w,b = self.param_tuples[-(i+1)]
            grads[w] = dw + 0.5*self.reg*2*self.params[w]
            grads[b] = db
            continue

        drelu = relu_backward(dx,relu_cache)
        dx,dw,db = affine_backward(drelu,affine_cache)




        w,b = self.param_tuples[-(i+1)]
        grads[w] = dw + 0.5*self.reg*2*self.params[w]
```

```
    grads[b] = db
#grads = {}

pass

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

This is the 2-layer neural network workbook for ECE 247 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 [1]:
```
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))))
```

Toy example

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

In [2]:
```
from nndl.neural_net import TwoLayerNet
```

In [3]:
```
# 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 [4]:
```
## 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:
3.381231248461569e-08
```

Forward pass loss

In [5]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

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

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

In [6]:
```
print(loss)
```

```
1.071696123862817
```

Backward pass

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

In [7]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward p
ass.
# If your implementation is correct, the difference between the numeric 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)

# 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], verbos
e=False)
    print('{} max relative error: {}'.format(param_name, rel_error(param_grad_
num, grads[param_name])))
```

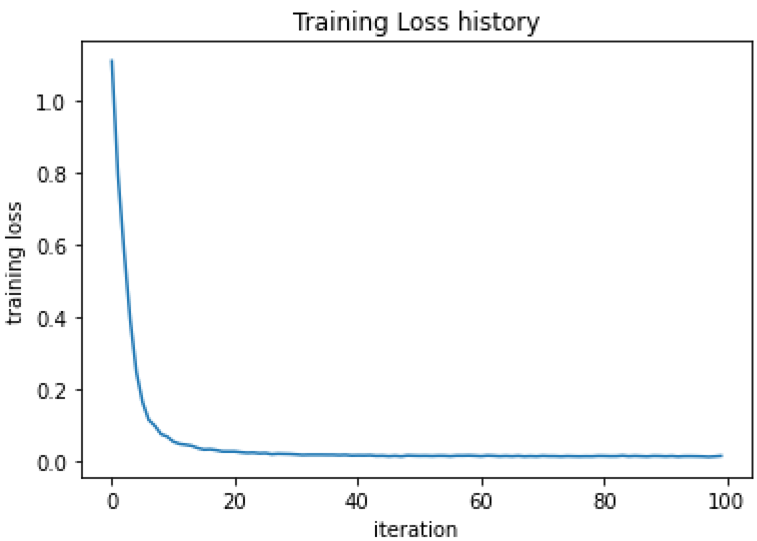
```
W1 max relative error: 1.2832892417669998e-09
W2 max relative error: 2.9632233460136427e-10
b1 max relative error: 3.172680285697327e-09
b2 max relative error: 1.2482624742512528e-09
```

Training the network

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

In [8]:
```
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:  0.014497864587765906
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [9]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = r'C:\Users\lpott\Desktop\UCLA\ECENGR247C-80\HW2\cifar-10-bat
ches-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 [10]:
```
input_size = 32 * 32 * 3
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=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# 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
```

```
iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.251825904316413
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897747
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

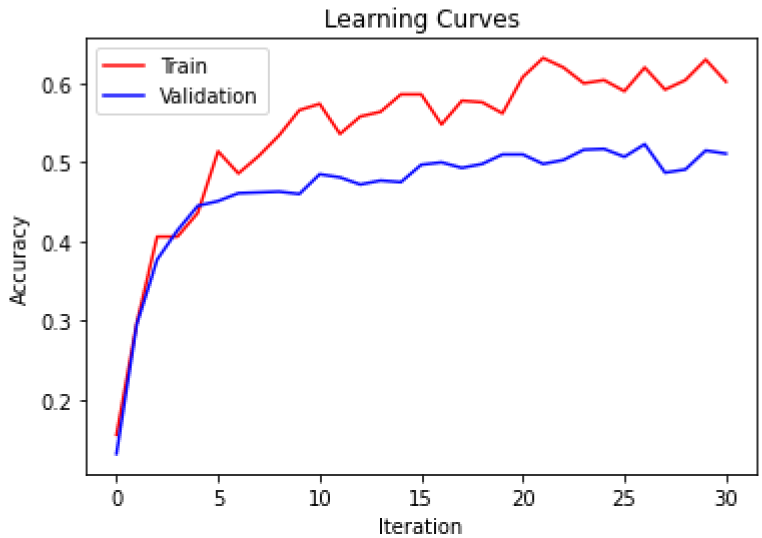
(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

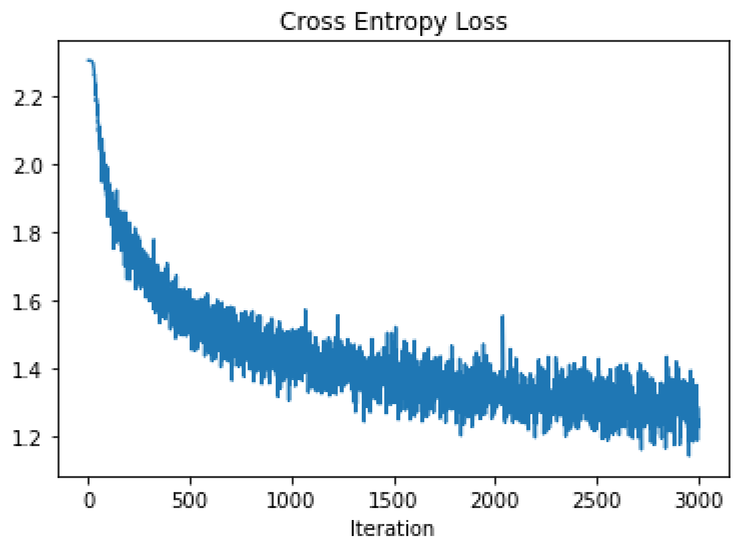
(2) How should you fix the problems you identified in (1)?

In [11]:
```
stats['train_acc_history']
```

Out[11]:
```
[0.095, 0.15, 0.25, 0.25, 0.315]
```

In [12]:
```
# ================================================================ #
# 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
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
batch_size = 500
learning_rate = 1e-3
learning_rate_decay = .99
reg = .2
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=3000, batch_size=batch_size,
            learning_rate=learning_rate, learning_rate_decay=learning_rate_dec
ay,
            reg=reg, verbose=False)

plt.plot(stats['train_acc_history'],'r-')
plt.plot(stats['val_acc_history'],'b-')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Learning Curves')
plt.legend(['Train','Validation'])

plt.figure()
plt.plot(stats['loss_history'])
plt.title('Cross Entropy Loss')
plt.xlabel('Iteration')
pass
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```





Answers:

(1) I noticed the trend that the regularization weight for W1 and W2 was too high, the batch size was too small, the learning rate was too small, the learning rate decay could be too high, and increasing the number of iterations helps.

(2) I should perform a grid-hyperparameter search to identify the optimal hyperparamer values for the batch size, learning rate, learning rate decay, and regularization weight.

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 [13]:
```
best_net = None # store the best model into this

# ================================================================ #
# 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)!
# ================================================================ #
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10

learning_rates = 1/np.logspace(3,6,num=10)
decay_rates = [.99,.95,.9,.5]
regularization = [0,0.05,.1,.2,.25,.5,.9,.99]
batch_sizes = [64,200,500]
best_acc = 0
for learning_rate in learning_rates:
    for learning_rate_decay in decay_rates:
        for reg in regularization:
            for batch_size in batch_sizes:
                net = TwoLayerNet(input_size, hidden_size, num_classes)

                # Train the network
                stats = net.train(X_train, y_train, X_val, y_val,
                            num_iters=3000, batch_size=batch_size,
                            learning_rate=learning_rate, learning_rate_decay=l
earning_rate_decay,
                            reg=reg, verbose=False)

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

                if best_acc < val_acc:
                    best_acc = val_acc
                    best_net = net
                pass

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
#best_net = net
```

```
0.001 0.99 0 64
Validation accuracy:  0.455
0.001 0.99 0 200
Validation accuracy:  0.519
0.001 0.99 0 500
Validation accuracy:  0.506
0.001 0.99 0.05 64
Validation accuracy:  0.454
0.001 0.99 0.05 200
Validation accuracy:  0.502
0.001 0.99 0.05 500
Validation accuracy:  0.51
0.001 0.99 0.1 64
Validation accuracy:  0.448
0.001 0.99 0.1 200
Validation accuracy:  0.494
0.001 0.99 0.1 500
Validation accuracy:  0.515
0.001 0.99 0.2 64
Validation accuracy:  0.479
0.001 0.99 0.2 200
Validation accuracy:  0.482
0.001 0.99 0.2 500
Validation accuracy:  0.493
0.001 0.99 0.25 64
Validation accuracy:  0.457
0.001 0.99 0.25 200
Validation accuracy:  0.516
0.001 0.99 0.25 500
Validation accuracy:  0.522
0.001 0.99 0.5 64
Validation accuracy:  0.468
0.001 0.99 0.5 200
Validation accuracy:  0.48
0.001 0.99 0.5 500
Validation accuracy:  0.494
0.001 0.99 0.9 64
Validation accuracy:  0.439
0.001 0.99 0.9 200
Validation accuracy:  0.482
0.001 0.99 0.9 500
Validation accuracy:  0.515
0.001 0.99 0.99 64
Validation accuracy:  0.428
0.001 0.99 0.99 200
Validation accuracy:  0.49
0.001 0.99 0.99 500
Validation accuracy:  0.496
0.001 0.95 0 64
Validation accuracy:  0.48
0.001 0.95 0 200
Validation accuracy:  0.512
0.001 0.95 0 500
Validation accuracy:  0.497
0.001 0.95 0.05 64
Validation accuracy:  0.449
0.001 0.95 0.05 200
```

```
Validation accuracy:  0.521
0.001 0.95 0.05 500
Validation accuracy:  0.512
0.001 0.95 0.1 64
Validation accuracy:  0.472
0.001 0.95 0.1 200
Validation accuracy:  0.51
0.001 0.95 0.1 500
Validation accuracy:  0.502
0.001 0.95 0.2 64
Validation accuracy:  0.462
0.001 0.95 0.2 200
Validation accuracy:  0.513
0.001 0.95 0.2 500
Validation accuracy:  0.51
0.001 0.95 0.25 64
Validation accuracy:  0.433
0.001 0.95 0.25 200
Validation accuracy:  0.505
0.001 0.95 0.25 500
Validation accuracy:  0.508
0.001 0.95 0.5 64
Validation accuracy:  0.481
0.001 0.95 0.5 200
Validation accuracy:  0.504
0.001 0.95 0.5 500
Validation accuracy:  0.509
0.001 0.95 0.9 64
Validation accuracy:  0.431
0.001 0.95 0.9 200
Validation accuracy:  0.498
0.001 0.95 0.9 500
Validation accuracy:  0.5
0.001 0.95 0.99 64
Validation accuracy:  0.444
0.001 0.95 0.99 200
Validation accuracy:  0.489
0.001 0.95 0.99 500
Validation accuracy:  0.492
0.001 0.9 0 64
Validation accuracy:  0.457
0.001 0.9 0 200
Validation accuracy:  0.515
0.001 0.9 0 500
Validation accuracy:  0.476
0.001 0.9 0.05 64
Validation accuracy:  0.447
0.001 0.9 0.05 200
Validation accuracy:  0.511
0.001 0.9 0.05 500
Validation accuracy:  0.483
0.001 0.9 0.1 64
Validation accuracy:  0.469
0.001 0.9 0.1 200
Validation accuracy:  0.508
0.001 0.9 0.1 500
Validation accuracy:  0.476
```

```
0.001 0.9 0.2 64
Validation accuracy:  0.451
0.001 0.9 0.2 200
Validation accuracy:  0.496
0.001 0.9 0.2 500
Validation accuracy:  0.497
0.001 0.9 0.25 64
Validation accuracy:  0.473
0.001 0.9 0.25 200
Validation accuracy:  0.507
0.001 0.9 0.25 500
Validation accuracy:  0.485
0.001 0.9 0.5 64
Validation accuracy:  0.462
0.001 0.9 0.5 200
Validation accuracy:  0.506
0.001 0.9 0.5 500
Validation accuracy:  0.492
0.001 0.9 0.9 64
Validation accuracy:  0.487
0.001 0.9 0.9 200
Validation accuracy:  0.491
0.001 0.9 0.9 500
Validation accuracy:  0.482
0.001 0.9 0.99 64
Validation accuracy:  0.431
0.001 0.9 0.99 200
Validation accuracy:  0.513
0.001 0.9 0.99 500
Validation accuracy:  0.483
0.001 0.5 0 64
Validation accuracy:  0.479
0.001 0.5 0 200
Validation accuracy:  0.396
0.001 0.5 0 500
Validation accuracy:  0.289
0.001 0.5 0.05 64
Validation accuracy:  0.489
0.001 0.5 0.05 200
Validation accuracy:  0.404
0.001 0.5 0.05 500
Validation accuracy:  0.291
0.001 0.5 0.1 64
Validation accuracy:  0.458
0.001 0.5 0.1 200
Validation accuracy:  0.401
0.001 0.5 0.1 500
Validation accuracy:  0.29
0.001 0.5 0.2 64
Validation accuracy:  0.466
0.001 0.5 0.2 200
Validation accuracy:  0.398
0.001 0.5 0.2 500
Validation accuracy:  0.301
0.001 0.5 0.25 64
Validation accuracy:  0.474
0.001 0.5 0.25 200
```

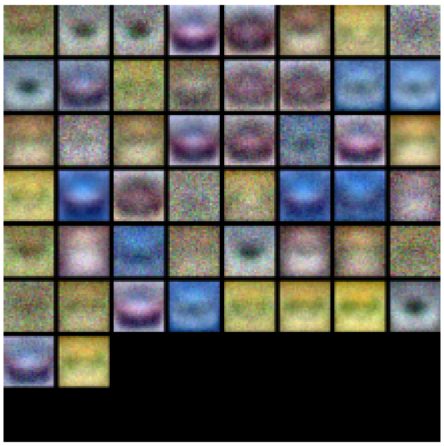
```
Validation accuracy:  0.413
0.001 0.5 0.25 500
Validation accuracy:  0.302
0.001 0.5 0.5 64
Validation accuracy:  0.45
0.001 0.5 0.5 200
Validation accuracy:  0.394
0.001 0.5 0.5 500
Validation accuracy:  0.306
0.001 0.5 0.9 64
Validation accuracy:  0.441
0.001 0.5 0.9 200
Validation accuracy:  0.397
0.001 0.5 0.9 500
Validation accuracy:  0.288
0.001 0.5 0.99 64
Validation accuracy:  0.478
0.001 0.5 0.99 200
Validation accuracy:  0.397
0.001 0.5 0.99 500
Validation accuracy:  0.285
0.0004641588833612777 0.99 0 64
Validation accuracy:  0.49
0.0004641588833612777 0.99 0 200
Validation accuracy:  0.484
0.0004641588833612777 0.99 0 500
Validation accuracy:  0.513
0.0004641588833612777 0.99 0.05 64
Validation accuracy:  0.488
0.0004641588833612777 0.99 0.05 200
Validation accuracy:  0.506
0.0004641588833612777 0.99 0.05 500
Validation accuracy:  0.492
0.0004641588833612777 0.99 0.1 64
Validation accuracy:  0.483
0.0004641588833612777 0.99 0.1 200
Validation accuracy:  0.502
0.0004641588833612777 0.99 0.1 500
Validation accuracy:  0.497
0.0004641588833612777 0.99 0.2 64
Validation accuracy:  0.491
0.0004641588833612777 0.99 0.2 200
Validation accuracy:  0.495
0.0004641588833612777 0.99 0.2 500
Validation accuracy:  0.502
0.0004641588833612777 0.99 0.25 64
Validation accuracy:  0.483
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-13-6ae0aa099f48> in <module>
     32                               num_iters=3000, batch_size=batch_size,
     33                               learning_rate=learning_rate, learning_rat
e_decay=learning_rate_decay,
---> 34                               reg=reg, verbose=False)
     35 
     36                 # Predict on the validation set

~\Desktop\UCLA\ECENGR247C-80\HW3-code\nndl\neural_net.py in train(self, X, y,
X_val, y_val, learning_rate, learning_rate_decay, reg, num_iters, batch_size,
verbose)
    191           # =========================================================
======= #
    192       idx_batch = np.random.choice(num_train,batch_size)
--> 193       X_batch = X[idx_batch,:]
    194       y_batch = y[idx_batch]
    195       pass

KeyboardInterrupt: 
```

In [14]:
```
print('Best Validation accuracy: ', best_acc)
```

```
Best Validation accuracy:  0.522
```

In [15]:
```
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) In the suboptimal net, I see some vague templates of objects such as a car, some blue blobs for ocean/sky, and some bright green blobs for grass (although mostly random pixel coloration), but for the best net that I arrived at templates are much more clear (a car is clearly visible, much more blue for ocean/sky, green appears more for grass, and less random pixel coloration with more solid color backgrounds).

Evaluate on test set

In [16]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.501
```

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

class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension 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 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

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

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

    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 neural
    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 each y[i] is
  an integer in the range 0 <= y[i] < C. This parameter is optional; if it
  is not passed then we only return scores, and if it is passed then 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.params.
"""
# 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 (N, C). 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.
# ================================================================ #
n1 = np.matmul(W1,X.T) + b1[:,np.newaxis]
l1 = np.clip(n1,a_min=0,a_max=float('inf'))
l2 = np.matmul(W2,l1) + b2[:,np.newaxis]
scores = l2.T
pass

# ================================================================ #
# 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 = None

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the loss of the neural network.  This includes the
#   softmax loss and the L2 regularization for W1 and W2. Store the
#   total loss in teh variable loss.  Multiply the regularization
#   loss by 0.5 (in addition to the factor reg).
```

```python
        # ================================================================ #
        reg_loss = 0.5 *(np.linalg.norm(W1)**2 + np.linalg.norm(W2)**2)
        softmax_loss = np.mean(-np.log(np.exp(scores[np.arange(N),y])/np.sum(np.exp(scores),1)))
        loss = softmax_loss + reg*reg_loss
        # scores is num_examples by num_classes
        pass
        # ================================================================ #
        # 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.
        # ================================================================ #
        Z = np.exp(scores)/np.sum(np.exp(scores),1)[:,np.newaxis]
        dLdz = np.copy(Z)
        dLdz[np.arange(N),y] = dLdz[np.arange(N),y] - 1
        dLdb2 = np.mean(dLdz,0).T
        dLdW2 = 1/N * np.matmul(dLdz.T,l1.T)
        dLdb1 = np.mean((n1>0)*np.matmul(W2.T,dLdz.T),1)
        dLdW1 = 1/N * np.matmul((n1>0)*np.matmul(W2.T,dLdz.T),X)

        grads['W1'] = dLdW1 + 0.5*reg*2*W1
        grads['W2'] = dLdW2 + 0.5*reg*2*W2
        grads['b1'] = dLdb1
        grads['b2'] = dLdb2

        pass

        # ================================================================ #
        # 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 means 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 learning 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.
  # ================================================================ #
  idx_batch = np.random.choice(num_train,batch_size)
  X_batch = X[idx_batch,:]
  y_batch = y[idx_batch]
  pass
  # ================================================================ #
  # 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 update
  #    all parameters (i.e., W1, W2, b1, and b2).
  # ================================================================ #
  self.params['W1'] -= learning_rate*grads['W1']
  self.params['W2'] -= learning_rate*grads['W2']

  self.params['b1'] -= learning_rate*grads['b1']
  self.params['b2'] -= learning_rate*grads['b2']

  pass

  # ================================================================ #
  # 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 rate.
  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 labels 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 points 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 predicted
          to have class c, where 0 <= c < C.
        """
        y_pred = None

        # ================================================================ #
        # YOUR CODE HERE:
        #   Predict the class given the input data.
        # ================================================================ #
        n1 = np.matmul(self.params['W1'],X.T) + self.params['b1'][:,np.newaxis]
        l1 = np.clip(n1,a_min=0,a_max=float('inf'))
        l2 = np.matmul(self.params['W2'],l1) + self.params['b2'][:,np.newaxis]
        y_pred = np.argmax(l2,0)

        pass


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

        return y_pred
```