In [1]:

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "Colab Notebooks/assignment1"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd drive/MyW Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/MyW Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/assignment1/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/assignment1
```

Fully-Connected Neural Nets

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

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

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

  return out, cache
```

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

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

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

```
  return dx, dw
```

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

In [16]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_a
from cs231n.solver import Solver

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

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

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

In [17]:
```
# Load the (preprocessed) CIFAR10 data.

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

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

Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

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

In [18]:
```
# Test the affine_forward function

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

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

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

```
b = np.linspace(-0.3, 0.1, num=output_dim)

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

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

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

Affine layer: backward

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

In [19]:

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

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

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

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

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

ReLU activation: forward

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

In [20]:

```
# Test the relu_forward function

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

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

# Compare your output with ours. The error should be on the order of e-8
```

```
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

ReLU activation: backward

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

In [21]:

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

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

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

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

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

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. Sigmoid
2. ReLU
3. Leaky ReLU

Answer: 1,2

Sigmoid 함수의 그라디언트는 input값이 매우 크거나 매우 작을 때 0이 되는 경향을 보인다

ReLU 함수의 경우 input값이 0보다 작을 때 그라디언트가 0이 된다.

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py` .

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

In [22]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

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

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

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

Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

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

In [25]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: 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 around the or
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

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

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be arour
print('₩nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09
```

```
Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

Two-layer network

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [28]:

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

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

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

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

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

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
```

```
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.21e-10
b1 relative error: 8.37e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. You also need to imeplement the `sgd` function in `cs231n/optim.py`. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

In [30]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

solver.train()


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

```
(Iteration 1 / 2450) loss: 2.301062
(Epoch 0 / 10) train acc: 0.115000; val_acc: 0.121000
(Iteration 11 / 2450) loss: 2.251480
(Iteration 21 / 2450) loss: 2.162527
(Iteration 31 / 2450) loss: 2.087260
(Iteration 41 / 2450) loss: 1.966636
(Iteration 51 / 2450) loss: 1.935143
(Iteration 61 / 2450) loss: 1.871565
(Iteration 71 / 2450) loss: 1.941934
(Iteration 81 / 2450) loss: 1.875348
(Iteration 91 / 2450) loss: 1.901880
```

```
(Iteration 101 / 2450) loss: 1.885025
(Iteration 111 / 2450) loss: 1.652062
(Iteration 121 / 2450) loss: 1.786294
(Iteration 131 / 2450) loss: 1.821498
(Iteration 141 / 2450) loss: 1.713176
(Iteration 151 / 2450) loss: 1.737538
(Iteration 161 / 2450) loss: 1.787771
(Iteration 171 / 2450) loss: 1.739175
(Iteration 181 / 2450) loss: 1.792364
(Iteration 191 / 2450) loss: 1.760089
(Iteration 201 / 2450) loss: 1.698475
(Iteration 211 / 2450) loss: 1.732730
(Iteration 221 / 2450) loss: 1.559826
(Iteration 231 / 2450) loss: 1.595973
(Iteration 241 / 2450) loss: 1.608602
(Epoch 1 / 10) train acc: 0.423000; val_acc: 0.420000
(Iteration 251 / 2450) loss: 1.579361
(Iteration 261 / 2450) loss: 1.651357
(Iteration 271 / 2450) loss: 1.575652
(Iteration 281 / 2450) loss: 1.772468
(Iteration 291 / 2450) loss: 1.582101
(Iteration 301 / 2450) loss: 1.519133
(Iteration 311 / 2450) loss: 1.622877
(Iteration 321 / 2450) loss: 1.684385
(Iteration 331 / 2450) loss: 1.657637
(Iteration 341 / 2450) loss: 1.622381
(Iteration 351 / 2450) loss: 1.582796
(Iteration 361 / 2450) loss: 1.561568
(Iteration 371 / 2450) loss: 1.631584
(Iteration 381 / 2450) loss: 1.482360
(Iteration 391 / 2450) loss: 1.568881
(Iteration 401 / 2450) loss: 1.599253
(Iteration 411 / 2450) loss: 1.533167
(Iteration 421 / 2450) loss: 1.577490
(Iteration 431 / 2450) loss: 1.471638
(Iteration 441 / 2450) loss: 1.551179
(Iteration 451 / 2450) loss: 1.483145
(Iteration 461 / 2450) loss: 1.477825
(Iteration 471 / 2450) loss: 1.459876
(Iteration 481 / 2450) loss: 1.513238
(Epoch 2 / 10) train acc: 0.427000; val_acc: 0.458000
(Iteration 491 / 2450) loss: 1.548793
(Iteration 501 / 2450) loss: 1.701152
(Iteration 511 / 2450) loss: 1.484739
(Iteration 521 / 2450) loss: 1.513854
(Iteration 531 / 2450) loss: 1.455637
(Iteration 541 / 2450) loss: 1.482078
(Iteration 551 / 2450) loss: 1.480574
(Iteration 561 / 2450) loss: 1.521879
(Iteration 571 / 2450) loss: 1.549874
(Iteration 581 / 2450) loss: 1.423816
(Iteration 591 / 2450) loss: 1.448922
(Iteration 601 / 2450) loss: 1.541726
(Iteration 611 / 2450) loss: 1.525935
(Iteration 621 / 2450) loss: 1.396273
(Iteration 631 / 2450) loss: 1.572838
(Iteration 641 / 2450) loss: 1.420059
(Iteration 651 / 2450) loss: 1.536649
(Iteration 661 / 2450) loss: 1.438662
(Iteration 671 / 2450) loss: 1.506135
(Iteration 681 / 2450) loss: 1.421433
(Iteration 691 / 2450) loss: 1.494946
(Iteration 701 / 2450) loss: 1.439626
(Iteration 711 / 2450) loss: 1.461976
(Iteration 721 / 2450) loss: 1.496525
(Iteration 731 / 2450) loss: 1.467832
(Epoch 3 / 10) train acc: 0.460000; val_acc: 0.489000
(Iteration 741 / 2450) loss: 1.436957
(Iteration 751 / 2450) loss: 1.524880
```

```
(Iteration 761 / 2450) loss: 1.442011
(Iteration 771 / 2450) loss: 1.434239
(Iteration 781 / 2450) loss: 1.367544
(Iteration 791 / 2450) loss: 1.390431
(Iteration 801 / 2450) loss: 1.313360
(Iteration 811 / 2450) loss: 1.648572
(Iteration 821 / 2450) loss: 1.421150
(Iteration 831 / 2450) loss: 1.548102
(Iteration 841 / 2450) loss: 1.370022
(Iteration 851 / 2450) loss: 1.302161
(Iteration 861 / 2450) loss: 1.480914
(Iteration 871 / 2450) loss: 1.514313
(Iteration 881 / 2450) loss: 1.364499
(Iteration 891 / 2450) loss: 1.395092
(Iteration 901 / 2450) loss: 1.546771
(Iteration 911 / 2450) loss: 1.303292
(Iteration 921 / 2450) loss: 1.349251
(Iteration 931 / 2450) loss: 1.471142
(Iteration 941 / 2450) loss: 1.521207
(Iteration 951 / 2450) loss: 1.448348
(Iteration 961 / 2450) loss: 1.309952
(Iteration 971 / 2450) loss: 1.506492
(Epoch 4 / 10) train acc: 0.505000; val_acc: 0.491000
(Iteration 981 / 2450) loss: 1.426253
(Iteration 991 / 2450) loss: 1.422465
(Iteration 1001 / 2450) loss: 1.411441
(Iteration 1011 / 2450) loss: 1.411264
(Iteration 1021 / 2450) loss: 1.415548
(Iteration 1031 / 2450) loss: 1.491388
(Iteration 1041 / 2450) loss: 1.435215
(Iteration 1051 / 2450) loss: 1.514544
(Iteration 1061 / 2450) loss: 1.499948
(Iteration 1071 / 2450) loss: 1.321867
(Iteration 1081 / 2450) loss: 1.311649
(Iteration 1091 / 2450) loss: 1.468143
(Iteration 1101 / 2450) loss: 1.422319
(Iteration 1111 / 2450) loss: 1.203739
(Iteration 1121 / 2450) loss: 1.434640
(Iteration 1131 / 2450) loss: 1.394254
(Iteration 1141 / 2450) loss: 1.442713
(Iteration 1151 / 2450) loss: 1.390939
(Iteration 1161 / 2450) loss: 1.313120
(Iteration 1171 / 2450) loss: 1.352683
(Iteration 1181 / 2450) loss: 1.304679
(Iteration 1191 / 2450) loss: 1.382310
(Iteration 1201 / 2450) loss: 1.441777
(Iteration 1211 / 2450) loss: 1.332536
(Iteration 1221 / 2450) loss: 1.533671
(Epoch 5 / 10) train acc: 0.507000; val_acc: 0.489000
(Iteration 1231 / 2450) loss: 1.370633
(Iteration 1241 / 2450) loss: 1.485948
(Iteration 1251 / 2450) loss: 1.419915
(Iteration 1261 / 2450) loss: 1.458735
(Iteration 1271 / 2450) loss: 1.387739
(Iteration 1281 / 2450) loss: 1.352125
(Iteration 1291 / 2450) loss: 1.468326
(Iteration 1301 / 2450) loss: 1.457903
(Iteration 1311 / 2450) loss: 1.196910
(Iteration 1321 / 2450) loss: 1.187926
(Iteration 1331 / 2450) loss: 1.464620
(Iteration 1341 / 2450) loss: 1.295607
(Iteration 1351 / 2450) loss: 1.350890
(Iteration 1361 / 2450) loss: 1.277080
(Iteration 1371 / 2450) loss: 1.382375
(Iteration 1381 / 2450) loss: 1.533723
(Iteration 1391 / 2450) loss: 1.357882
(Iteration 1401 / 2450) loss: 1.197675
(Iteration 1411 / 2450) loss: 1.335544
(Iteration 1421 / 2450) loss: 1.333597
```

```
(Iteration 1431 / 2450) loss: 1.327684
(Iteration 1441 / 2450) loss: 1.370115
(Iteration 1451 / 2450) loss: 1.315832
(Iteration 1461 / 2450) loss: 1.447350
(Epoch 6 / 10) train acc: 0.539000; val_acc: 0.502000
(Iteration 1471 / 2450) loss: 1.495985
(Iteration 1481 / 2450) loss: 1.387942
(Iteration 1491 / 2450) loss: 1.314551
(Iteration 1501 / 2450) loss: 1.419959
(Iteration 1511 / 2450) loss: 1.405969
(Iteration 1521 / 2450) loss: 1.442581
(Iteration 1531 / 2450) loss: 1.350303
(Iteration 1541 / 2450) loss: 1.327611
(Iteration 1551 / 2450) loss: 1.251457
(Iteration 1561 / 2450) loss: 1.271578
(Iteration 1571 / 2450) loss: 1.369314
(Iteration 1581 / 2450) loss: 1.236262
(Iteration 1591 / 2450) loss: 1.322605
(Iteration 1601 / 2450) loss: 1.369856
(Iteration 1611 / 2450) loss: 1.215187
(Iteration 1621 / 2450) loss: 1.342558
(Iteration 1631 / 2450) loss: 1.435601
(Iteration 1641 / 2450) loss: 1.250940
(Iteration 1651 / 2450) loss: 1.316388
(Iteration 1661 / 2450) loss: 1.298129
(Iteration 1671 / 2450) loss: 1.269967
(Iteration 1681 / 2450) loss: 1.491559
(Iteration 1691 / 2450) loss: 1.208586
(Iteration 1701 / 2450) loss: 1.304255
(Iteration 1711 / 2450) loss: 1.382717
(Epoch 7 / 10) train acc: 0.514000; val_acc: 0.498000
(Iteration 1721 / 2450) loss: 1.409847
(Iteration 1731 / 2450) loss: 1.318587
(Iteration 1741 / 2450) loss: 1.143294
(Iteration 1751 / 2450) loss: 1.286489
(Iteration 1761 / 2450) loss: 1.328778
(Iteration 1771 / 2450) loss: 1.319623
(Iteration 1781 / 2450) loss: 1.411528
(Iteration 1791 / 2450) loss: 1.504018
(Iteration 1801 / 2450) loss: 1.458307
(Iteration 1811 / 2450) loss: 1.301192
(Iteration 1821 / 2450) loss: 1.150706
(Iteration 1831 / 2450) loss: 1.344912
(Iteration 1841 / 2450) loss: 1.309328
(Iteration 1851 / 2450) loss: 1.306356
(Iteration 1861 / 2450) loss: 1.285387
(Iteration 1871 / 2450) loss: 1.222052
(Iteration 1881 / 2450) loss: 1.307119
(Iteration 1891 / 2450) loss: 1.281371
(Iteration 1901 / 2450) loss: 1.317703
(Iteration 1911 / 2450) loss: 1.268661
(Iteration 1921 / 2450) loss: 1.293358
(Iteration 1931 / 2450) loss: 1.261006
(Iteration 1941 / 2450) loss: 1.242547
(Iteration 1951 / 2450) loss: 1.238660
(Epoch 8 / 10) train acc: 0.542000; val_acc: 0.514000
(Iteration 1961 / 2450) loss: 1.301729
(Iteration 1971 / 2450) loss: 1.227864
(Iteration 1981 / 2450) loss: 1.400188
(Iteration 1991 / 2450) loss: 1.294325
(Iteration 2001 / 2450) loss: 1.231971
(Iteration 2011 / 2450) loss: 1.251470
(Iteration 2021 / 2450) loss: 1.242564
(Iteration 2031 / 2450) loss: 1.373022
(Iteration 2041 / 2450) loss: 1.307488
(Iteration 2051 / 2450) loss: 1.276249
(Iteration 2061 / 2450) loss: 1.248470
(Iteration 2071 / 2450) loss: 1.337349
(Iteration 2081 / 2450) loss: 1.322487
```

```
(Iteration 2091 / 2450) loss: 1.246377
(Iteration 2101 / 2450) loss: 1.193440
(Iteration 2111 / 2450) loss: 1.311307
(Iteration 2121 / 2450) loss: 1.205550
(Iteration 2131 / 2450) loss: 1.257572
(Iteration 2141 / 2450) loss: 1.147793
(Iteration 2151 / 2450) loss: 1.395807
(Iteration 2161 / 2450) loss: 1.250263
(Iteration 2171 / 2450) loss: 1.404383
(Iteration 2181 / 2450) loss: 1.259556
(Iteration 2191 / 2450) loss: 1.488411
(Iteration 2201 / 2450) loss: 1.172424
(Epoch 9 / 10) train acc: 0.517000; val_acc: 0.505000
(Iteration 2211 / 2450) loss: 1.279431
(Iteration 2221 / 2450) loss: 1.209400
(Iteration 2231 / 2450) loss: 1.391287
(Iteration 2241 / 2450) loss: 1.366110
(Iteration 2251 / 2450) loss: 1.244494
(Iteration 2261 / 2450) loss: 1.312080
(Iteration 2271 / 2450) loss: 1.205155
(Iteration 2281 / 2450) loss: 1.441035
(Iteration 2291 / 2450) loss: 1.354273
(Iteration 2301 / 2450) loss: 1.235923
(Iteration 2311 / 2450) loss: 1.292529
(Iteration 2321 / 2450) loss: 1.247569
(Iteration 2331 / 2450) loss: 1.344386
(Iteration 2341 / 2450) loss: 1.275198
(Iteration 2351 / 2450) loss: 1.266819
(Iteration 2361 / 2450) loss: 1.356471
(Iteration 2371 / 2450) loss: 1.290343
(Iteration 2381 / 2450) loss: 1.159217
(Iteration 2391 / 2450) loss: 1.369927
(Iteration 2401 / 2450) loss: 1.229884
(Iteration 2411 / 2450) loss: 1.394268
(Iteration 2421 / 2450) loss: 1.374488
(Iteration 2431 / 2450) loss: 1.200501
(Iteration 2441 / 2450) loss: 1.351379
(Epoch 10 / 10) train acc: 0.555000; val_acc: 0.506000
```

Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

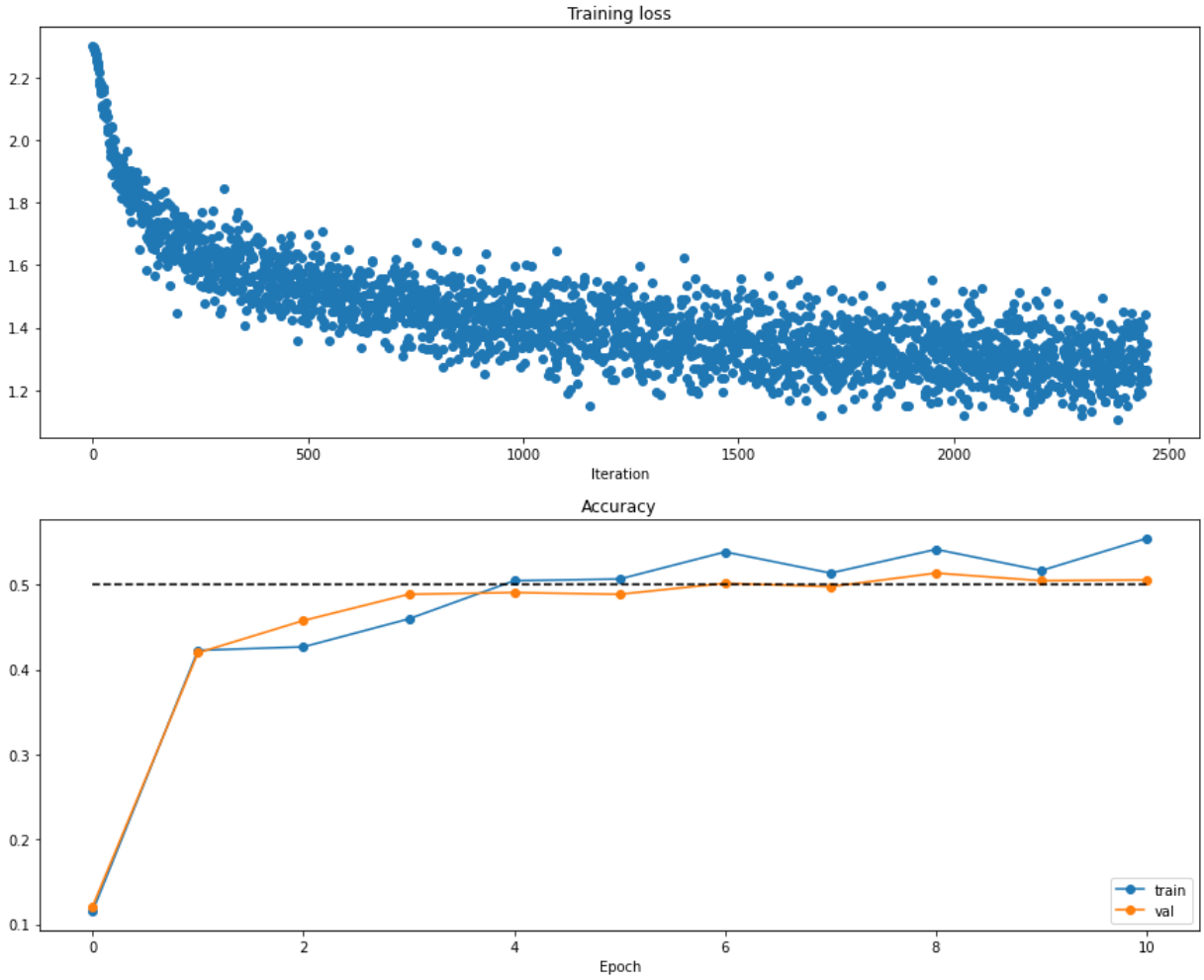
In [31]:

```
# 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()
```



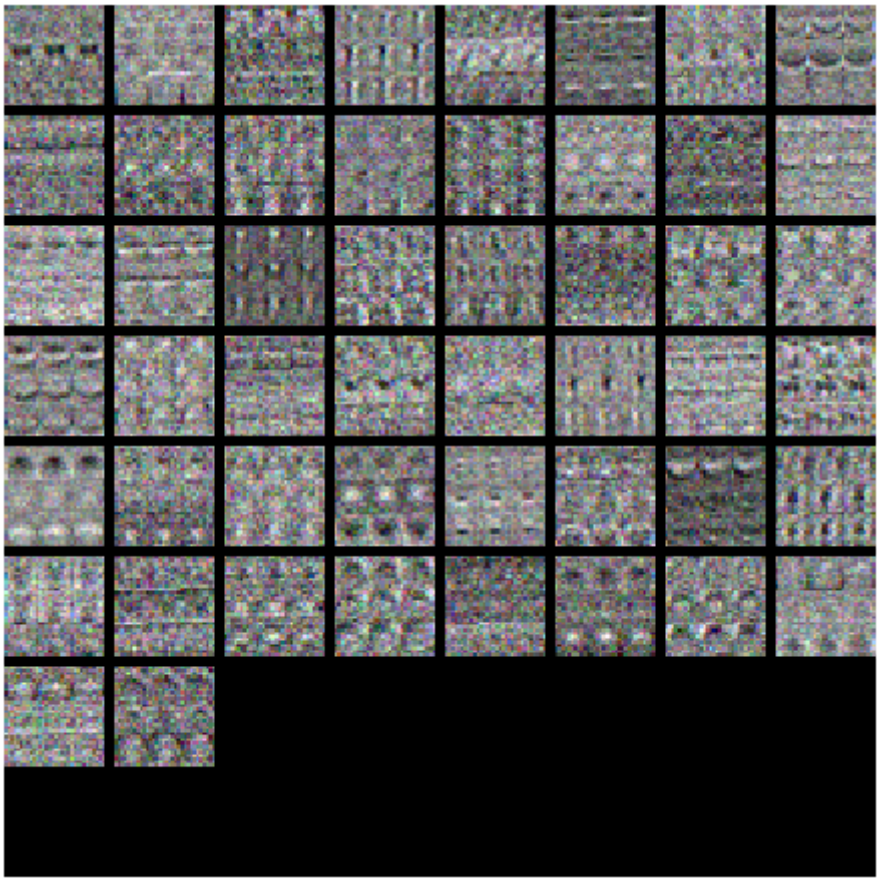
In [32]:

```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.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(model)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [46]:

```
best_model = None
best_acc = 0
```

```
################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_model.                                                          #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on thexs previous exercises.                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

from itertools import product

input_sizes = [32 * 32 * 3]
hidden_sizes = [30,50,70] # 작업용이 아니고, 연습용이므로 매우 간단하게만 하겠습니다.
num_classes = [10]

grid_search = list(product(*[input_sizes, hidden_sizes, num_classes]))

for input_size, hidden_size, num_class in grid_search :
  model = TwoLayerNet(input_size, hidden_size, num_class)


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

  solver.train()
  if best_acc < solver.val_acc_history[-1]:
    best_model = model

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

```
(Iteration 1 / 2450) loss: 2.301240
(Epoch 0 / 10) train acc: 0.136000; val_acc: 0.146000
(Iteration 11 / 2450) loss: 2.277555
(Iteration 21 / 2450) loss: 2.229510
(Iteration 31 / 2450) loss: 2.101831
(Iteration 41 / 2450) loss: 2.012400
(Iteration 51 / 2450) loss: 2.041533
(Iteration 61 / 2450) loss: 1.988597
(Iteration 71 / 2450) loss: 1.849398
(Iteration 81 / 2450) loss: 1.879167
(Iteration 91 / 2450) loss: 1.827818
(Iteration 101 / 2450) loss: 1.910156
(Iteration 111 / 2450) loss: 1.823634
(Iteration 121 / 2450) loss: 1.706116
(Iteration 131 / 2450) loss: 1.947344
(Iteration 141 / 2450) loss: 1.770316
(Iteration 151 / 2450) loss: 1.814911
(Iteration 161 / 2450) loss: 1.836452
(Iteration 171 / 2450) loss: 1.765615
(Iteration 181 / 2450) loss: 1.774654
(Iteration 191 / 2450) loss: 1.794811
(Iteration 201 / 2450) loss: 1.823110
(Iteration 211 / 2450) loss: 1.579126
(Iteration 221 / 2450) loss: 1.727079
```

```
(Iteration 231 / 2450) loss: 1.745523
(Iteration 241 / 2450) loss: 1.645598
(Epoch 1 / 10) train acc: 0.428000; val_acc: 0.421000
(Iteration 251 / 2450) loss: 1.636971
(Iteration 261 / 2450) loss: 1.603092
(Iteration 271 / 2450) loss: 1.688406
(Iteration 281 / 2450) loss: 1.510255
(Iteration 291 / 2450) loss: 1.762342
(Iteration 301 / 2450) loss: 1.703611
(Iteration 311 / 2450) loss: 1.679737
(Iteration 321 / 2450) loss: 1.665730
(Iteration 331 / 2450) loss: 1.674120
(Iteration 341 / 2450) loss: 1.629407
(Iteration 351 / 2450) loss: 1.551912
(Iteration 361 / 2450) loss: 1.615380
(Iteration 371 / 2450) loss: 1.543083
(Iteration 381 / 2450) loss: 1.605738
(Iteration 391 / 2450) loss: 1.505873
(Iteration 401 / 2450) loss: 1.542002
(Iteration 411 / 2450) loss: 1.732901
(Iteration 421 / 2450) loss: 1.605006
(Iteration 431 / 2450) loss: 1.600134
(Iteration 441 / 2450) loss: 1.539106
(Iteration 451 / 2450) loss: 1.557983
(Iteration 461 / 2450) loss: 1.549422
(Iteration 471 / 2450) loss: 1.657690
(Iteration 481 / 2450) loss: 1.702324
(Epoch 2 / 10) train acc: 0.454000; val_acc: 0.453000
(Iteration 491 / 2450) loss: 1.627886
(Iteration 501 / 2450) loss: 1.618485
(Iteration 511 / 2450) loss: 1.629024
(Iteration 521 / 2450) loss: 1.554362
(Iteration 531 / 2450) loss: 1.538928
(Iteration 541 / 2450) loss: 1.648469
(Iteration 551 / 2450) loss: 1.658718
(Iteration 561 / 2450) loss: 1.449315
(Iteration 571 / 2450) loss: 1.606690
(Iteration 581 / 2450) loss: 1.602823
(Iteration 591 / 2450) loss: 1.535803
(Iteration 601 / 2450) loss: 1.595409
(Iteration 611 / 2450) loss: 1.525669
(Iteration 621 / 2450) loss: 1.645790
(Iteration 631 / 2450) loss: 1.597733
(Iteration 641 / 2450) loss: 1.599922
(Iteration 651 / 2450) loss: 1.569058
(Iteration 661 / 2450) loss: 1.442391
(Iteration 671 / 2450) loss: 1.547918
(Iteration 681 / 2450) loss: 1.608452
(Iteration 691 / 2450) loss: 1.544600
(Iteration 701 / 2450) loss: 1.575980
(Iteration 711 / 2450) loss: 1.550880
(Iteration 721 / 2450) loss: 1.589846
(Iteration 731 / 2450) loss: 1.644839
(Epoch 3 / 10) train acc: 0.477000; val_acc: 0.462000
(Iteration 741 / 2450) loss: 1.491753
(Iteration 751 / 2450) loss: 1.430144
(Iteration 761 / 2450) loss: 1.478200
(Iteration 771 / 2450) loss: 1.614174
(Iteration 781 / 2450) loss: 1.369998
(Iteration 791 / 2450) loss: 1.510988
(Iteration 801 / 2450) loss: 1.413751
(Iteration 811 / 2450) loss: 1.450057
(Iteration 821 / 2450) loss: 1.530228
(Iteration 831 / 2450) loss: 1.508333
(Iteration 841 / 2450) loss: 1.487894
(Iteration 851 / 2450) loss: 1.431422
(Iteration 861 / 2450) loss: 1.570914
(Iteration 871 / 2450) loss: 1.588590
(Iteration 881 / 2450) loss: 1.467768
```

```
(Iteration 891 / 2450) loss: 1.520918
(Iteration 901 / 2450) loss: 1.333443
(Iteration 911 / 2450) loss: 1.467762
(Iteration 921 / 2450) loss: 1.606175
(Iteration 931 / 2450) loss: 1.593782
(Iteration 941 / 2450) loss: 1.382505
(Iteration 951 / 2450) loss: 1.437738
(Iteration 961 / 2450) loss: 1.537754
(Iteration 971 / 2450) loss: 1.566210
(Epoch 4 / 10) train acc: 0.472000; val_acc: 0.473000
(Iteration 981 / 2450) loss: 1.528383
(Iteration 991 / 2450) loss: 1.416630
(Iteration 1001 / 2450) loss: 1.542181
(Iteration 1011 / 2450) loss: 1.432989
(Iteration 1021 / 2450) loss: 1.431664
(Iteration 1031 / 2450) loss: 1.498961
(Iteration 1041 / 2450) loss: 1.523007
(Iteration 1051 / 2450) loss: 1.392440
(Iteration 1061 / 2450) loss: 1.284117
(Iteration 1071 / 2450) loss: 1.486153
(Iteration 1081 / 2450) loss: 1.389763
(Iteration 1091 / 2450) loss: 1.454384
(Iteration 1101 / 2450) loss: 1.477839
(Iteration 1111 / 2450) loss: 1.510576
(Iteration 1121 / 2450) loss: 1.376945
(Iteration 1131 / 2450) loss: 1.401995
(Iteration 1141 / 2450) loss: 1.519880
(Iteration 1151 / 2450) loss: 1.362724
(Iteration 1161 / 2450) loss: 1.420891
(Iteration 1171 / 2450) loss: 1.596354
(Iteration 1181 / 2450) loss: 1.320220
(Iteration 1191 / 2450) loss: 1.338381
(Iteration 1201 / 2450) loss: 1.378808
(Iteration 1211 / 2450) loss: 1.416045
(Iteration 1221 / 2450) loss: 1.372824
(Epoch 5 / 10) train acc: 0.525000; val_acc: 0.477000
(Iteration 1231 / 2450) loss: 1.397863
(Iteration 1241 / 2450) loss: 1.299055
(Iteration 1251 / 2450) loss: 1.480183
(Iteration 1261 / 2450) loss: 1.432729
(Iteration 1271 / 2450) loss: 1.409512
(Iteration 1281 / 2450) loss: 1.569342
(Iteration 1291 / 2450) loss: 1.401750
(Iteration 1301 / 2450) loss: 1.381314
(Iteration 1311 / 2450) loss: 1.461025
(Iteration 1321 / 2450) loss: 1.393709
(Iteration 1331 / 2450) loss: 1.488881
(Iteration 1341 / 2450) loss: 1.507709
(Iteration 1351 / 2450) loss: 1.455347
(Iteration 1361 / 2450) loss: 1.561728
(Iteration 1371 / 2450) loss: 1.374803
(Iteration 1381 / 2450) loss: 1.393230
(Iteration 1391 / 2450) loss: 1.453002
(Iteration 1401 / 2450) loss: 1.409842
(Iteration 1411 / 2450) loss: 1.446954
(Iteration 1421 / 2450) loss: 1.343915
(Iteration 1431 / 2450) loss: 1.577752
(Iteration 1441 / 2450) loss: 1.422560
(Iteration 1451 / 2450) loss: 1.445700
(Iteration 1461 / 2450) loss: 1.446991
(Epoch 6 / 10) train acc: 0.493000; val_acc: 0.464000
(Iteration 1471 / 2450) loss: 1.479682
(Iteration 1481 / 2450) loss: 1.585364
(Iteration 1491 / 2450) loss: 1.395088
(Iteration 1501 / 2450) loss: 1.312716
(Iteration 1511 / 2450) loss: 1.430084
(Iteration 1521 / 2450) loss: 1.431522
(Iteration 1531 / 2450) loss: 1.254516
(Iteration 1541 / 2450) loss: 1.362921
```

```
(Iteration 1551 / 2450) loss: 1.373446
(Iteration 1561 / 2450) loss: 1.318716
(Iteration 1571 / 2450) loss: 1.480803
(Iteration 1581 / 2450) loss: 1.422848
(Iteration 1591 / 2450) loss: 1.524721
(Iteration 1601 / 2450) loss: 1.417036
(Iteration 1611 / 2450) loss: 1.430313
(Iteration 1621 / 2450) loss: 1.414675
(Iteration 1631 / 2450) loss: 1.387570
(Iteration 1641 / 2450) loss: 1.533697
(Iteration 1651 / 2450) loss: 1.486892
(Iteration 1661 / 2450) loss: 1.411898
(Iteration 1671 / 2450) loss: 1.463555
(Iteration 1681 / 2450) loss: 1.483090
(Iteration 1691 / 2450) loss: 1.509243
(Iteration 1701 / 2450) loss: 1.526401
(Iteration 1711 / 2450) loss: 1.394608
(Epoch 7 / 10) train acc: 0.512000; val_acc: 0.472000
(Iteration 1721 / 2450) loss: 1.381666
(Iteration 1731 / 2450) loss: 1.515665
(Iteration 1741 / 2450) loss: 1.309472
(Iteration 1751 / 2450) loss: 1.324339
(Iteration 1761 / 2450) loss: 1.400291
(Iteration 1771 / 2450) loss: 1.327295
(Iteration 1781 / 2450) loss: 1.397114
(Iteration 1791 / 2450) loss: 1.531224
(Iteration 1801 / 2450) loss: 1.466893
(Iteration 1811 / 2450) loss: 1.272893
(Iteration 1821 / 2450) loss: 1.486047
(Iteration 1831 / 2450) loss: 1.275141
(Iteration 1841 / 2450) loss: 1.440627
(Iteration 1851 / 2450) loss: 1.538354
(Iteration 1861 / 2450) loss: 1.523676
(Iteration 1871 / 2450) loss: 1.417632
(Iteration 1881 / 2450) loss: 1.354648
(Iteration 1891 / 2450) loss: 1.406255
(Iteration 1901 / 2450) loss: 1.343739
(Iteration 1911 / 2450) loss: 1.516996
(Iteration 1921 / 2450) loss: 1.575259
(Iteration 1931 / 2450) loss: 1.395426
(Iteration 1941 / 2450) loss: 1.390626
(Iteration 1951 / 2450) loss: 1.301946
(Epoch 8 / 10) train acc: 0.520000; val_acc: 0.478000
(Iteration 1961 / 2450) loss: 1.427316
(Iteration 1971 / 2450) loss: 1.360132
(Iteration 1981 / 2450) loss: 1.352709
(Iteration 1991 / 2450) loss: 1.389088
(Iteration 2001 / 2450) loss: 1.457409
(Iteration 2011 / 2450) loss: 1.359508
(Iteration 2021 / 2450) loss: 1.507706
(Iteration 2031 / 2450) loss: 1.301192
(Iteration 2041 / 2450) loss: 1.266537
(Iteration 2051 / 2450) loss: 1.589513
(Iteration 2061 / 2450) loss: 1.564536
(Iteration 2071 / 2450) loss: 1.387688
(Iteration 2081 / 2450) loss: 1.576339
(Iteration 2091 / 2450) loss: 1.300779
(Iteration 2101 / 2450) loss: 1.381166
(Iteration 2111 / 2450) loss: 1.483371
(Iteration 2121 / 2450) loss: 1.251985
(Iteration 2131 / 2450) loss: 1.493326
(Iteration 2141 / 2450) loss: 1.404395
(Iteration 2151 / 2450) loss: 1.258996
(Iteration 2161 / 2450) loss: 1.315686
(Iteration 2171 / 2450) loss: 1.515904
(Iteration 2181 / 2450) loss: 1.328388
(Iteration 2191 / 2450) loss: 1.386794
(Iteration 2201 / 2450) loss: 1.423012
(Epoch 9 / 10) train acc: 0.525000; val_acc: 0.482000
```

```
(Iteration 2211 / 2450) loss: 1.345283
(Iteration 2221 / 2450) loss: 1.368190
(Iteration 2231 / 2450) loss: 1.358666
(Iteration 2241 / 2450) loss: 1.351290
(Iteration 2251 / 2450) loss: 1.423168
(Iteration 2261 / 2450) loss: 1.346814
(Iteration 2271 / 2450) loss: 1.393914
(Iteration 2281 / 2450) loss: 1.360862
(Iteration 2291 / 2450) loss: 1.338820
(Iteration 2301 / 2450) loss: 1.476050
(Iteration 2311 / 2450) loss: 1.334639
(Iteration 2321 / 2450) loss: 1.407816
(Iteration 2331 / 2450) loss: 1.353286
(Iteration 2341 / 2450) loss: 1.478509
(Iteration 2351 / 2450) loss: 1.378958
(Iteration 2361 / 2450) loss: 1.393050
(Iteration 2371 / 2450) loss: 1.289774
(Iteration 2381 / 2450) loss: 1.265370
(Iteration 2391 / 2450) loss: 1.357864
(Iteration 2401 / 2450) loss: 1.250289
(Iteration 2411 / 2450) loss: 1.382425
(Iteration 2421 / 2450) loss: 1.345239
(Iteration 2431 / 2450) loss: 1.476200
(Iteration 2441 / 2450) loss: 1.400283
(Epoch 10 / 10) train acc: 0.525000; val_acc: 0.495000
(Iteration 1 / 2450) loss: 2.304754
(Epoch 0 / 10) train acc: 0.116000; val_acc: 0.124000
(Iteration 11 / 2450) loss: 2.268288
(Iteration 21 / 2450) loss: 2.188824
(Iteration 31 / 2450) loss: 2.095697
(Iteration 41 / 2450) loss: 2.044836
(Iteration 51 / 2450) loss: 1.987263
(Iteration 61 / 2450) loss: 1.940641
(Iteration 71 / 2450) loss: 1.854705
(Iteration 81 / 2450) loss: 1.924250
(Iteration 91 / 2450) loss: 1.817332
(Iteration 101 / 2450) loss: 1.815290
(Iteration 111 / 2450) loss: 1.832897
(Iteration 121 / 2450) loss: 1.768347
(Iteration 131 / 2450) loss: 1.672730
(Iteration 141 / 2450) loss: 1.776624
(Iteration 151 / 2450) loss: 1.631340
(Iteration 161 / 2450) loss: 1.706623
(Iteration 171 / 2450) loss: 1.703016
(Iteration 181 / 2450) loss: 1.815293
(Iteration 191 / 2450) loss: 1.880385
(Iteration 201 / 2450) loss: 1.695288
(Iteration 211 / 2450) loss: 1.655210
(Iteration 221 / 2450) loss: 1.648432
(Iteration 231 / 2450) loss: 1.636977
(Iteration 241 / 2450) loss: 1.572576
(Epoch 1 / 10) train acc: 0.393000; val_acc: 0.395000
(Iteration 251 / 2450) loss: 1.757195
(Iteration 261 / 2450) loss: 1.460157
(Iteration 271 / 2450) loss: 1.663241
(Iteration 281 / 2450) loss: 1.677534
(Iteration 291 / 2450) loss: 1.689579
(Iteration 301 / 2450) loss: 1.726867
(Iteration 311 / 2450) loss: 1.617059
(Iteration 321 / 2450) loss: 1.598061
(Iteration 331 / 2450) loss: 1.759665
(Iteration 341 / 2450) loss: 1.716363
(Iteration 351 / 2450) loss: 1.554991
(Iteration 361 / 2450) loss: 1.502879
(Iteration 371 / 2450) loss: 1.645517
(Iteration 381 / 2450) loss: 1.578636
(Iteration 391 / 2450) loss: 1.446598
(Iteration 401 / 2450) loss: 1.673109
(Iteration 411 / 2450) loss: 1.511414
```

```
(Iteration 421 / 2450) loss: 1.423655
(Iteration 431 / 2450) loss: 1.467815
(Iteration 441 / 2450) loss: 1.561064
(Iteration 451 / 2450) loss: 1.566624
(Iteration 461 / 2450) loss: 1.457665
(Iteration 471 / 2450) loss: 1.413555
(Iteration 481 / 2450) loss: 1.530215
(Epoch 2 / 10) train acc: 0.473000; val_acc: 0.452000
(Iteration 491 / 2450) loss: 1.625060
(Iteration 501 / 2450) loss: 1.591361
(Iteration 511 / 2450) loss: 1.555514
(Iteration 521 / 2450) loss: 1.468932
(Iteration 531 / 2450) loss: 1.544481
(Iteration 541 / 2450) loss: 1.420925
(Iteration 551 / 2450) loss: 1.503597
(Iteration 561 / 2450) loss: 1.482064
(Iteration 571 / 2450) loss: 1.485845
(Iteration 581 / 2450) loss: 1.389664
(Iteration 591 / 2450) loss: 1.491425
(Iteration 601 / 2450) loss: 1.418952
(Iteration 611 / 2450) loss: 1.421455
(Iteration 621 / 2450) loss: 1.530390
(Iteration 631 / 2450) loss: 1.465403
(Iteration 641 / 2450) loss: 1.552890
(Iteration 651 / 2450) loss: 1.502045
(Iteration 661 / 2450) loss: 1.441957
(Iteration 671 / 2450) loss: 1.459177
(Iteration 681 / 2450) loss: 1.457458
(Iteration 691 / 2450) loss: 1.428370
(Iteration 701 / 2450) loss: 1.661617
(Iteration 711 / 2450) loss: 1.497450
(Iteration 721 / 2450) loss: 1.565460
(Iteration 731 / 2450) loss: 1.430234
(Epoch 3 / 10) train acc: 0.466000; val_acc: 0.482000
(Iteration 741 / 2450) loss: 1.528696
(Iteration 751 / 2450) loss: 1.352447
(Iteration 761 / 2450) loss: 1.436820
(Iteration 771 / 2450) loss: 1.469715
(Iteration 781 / 2450) loss: 1.595534
(Iteration 791 / 2450) loss: 1.511609
(Iteration 801 / 2450) loss: 1.456536
(Iteration 811 / 2450) loss: 1.416786
(Iteration 821 / 2450) loss: 1.572525
(Iteration 831 / 2450) loss: 1.503684
(Iteration 841 / 2450) loss: 1.470118
(Iteration 851 / 2450) loss: 1.400845
(Iteration 861 / 2450) loss: 1.415821
(Iteration 871 / 2450) loss: 1.458609
(Iteration 881 / 2450) loss: 1.520684
(Iteration 891 / 2450) loss: 1.556430
(Iteration 901 / 2450) loss: 1.440043
(Iteration 911 / 2450) loss: 1.565673
(Iteration 921 / 2450) loss: 1.410106
(Iteration 931 / 2450) loss: 1.347095
(Iteration 941 / 2450) loss: 1.411653
(Iteration 951 / 2450) loss: 1.510119
(Iteration 961 / 2450) loss: 1.358242
(Iteration 971 / 2450) loss: 1.427202
(Epoch 4 / 10) train acc: 0.505000; val_acc: 0.505000
(Iteration 981 / 2450) loss: 1.400222
(Iteration 991 / 2450) loss: 1.337681
(Iteration 1001 / 2450) loss: 1.364654
(Iteration 1011 / 2450) loss: 1.434066
(Iteration 1021 / 2450) loss: 1.419782
(Iteration 1031 / 2450) loss: 1.373067
(Iteration 1041 / 2450) loss: 1.515980
(Iteration 1051 / 2450) loss: 1.403025
(Iteration 1061 / 2450) loss: 1.449963
(Iteration 1071 / 2450) loss: 1.369264
```

```
(Iteration 1081 / 2450) loss: 1.426192
(Iteration 1091 / 2450) loss: 1.461990
(Iteration 1101 / 2450) loss: 1.309483
(Iteration 1111 / 2450) loss: 1.361956
(Iteration 1121 / 2450) loss: 1.397992
(Iteration 1131 / 2450) loss: 1.436921
(Iteration 1141 / 2450) loss: 1.338712
(Iteration 1151 / 2450) loss: 1.307334
(Iteration 1161 / 2450) loss: 1.378457
(Iteration 1171 / 2450) loss: 1.348299
(Iteration 1181 / 2450) loss: 1.351242
(Iteration 1191 / 2450) loss: 1.543076
(Iteration 1201 / 2450) loss: 1.343419
(Iteration 1211 / 2450) loss: 1.439975
(Iteration 1221 / 2450) loss: 1.399033
(Epoch 5 / 10) train acc: 0.542000; val_acc: 0.514000
(Iteration 1231 / 2450) loss: 1.378103
(Iteration 1241 / 2450) loss: 1.501666
(Iteration 1251 / 2450) loss: 1.468319
(Iteration 1261 / 2450) loss: 1.528408
(Iteration 1271 / 2450) loss: 1.375442
(Iteration 1281 / 2450) loss: 1.402498
(Iteration 1291 / 2450) loss: 1.404433
(Iteration 1301 / 2450) loss: 1.451954
(Iteration 1311 / 2450) loss: 1.377714
(Iteration 1321 / 2450) loss: 1.427450
(Iteration 1331 / 2450) loss: 1.417703
(Iteration 1341 / 2450) loss: 1.370502
(Iteration 1351 / 2450) loss: 1.283328
(Iteration 1361 / 2450) loss: 1.248631
(Iteration 1371 / 2450) loss: 1.277504
(Iteration 1381 / 2450) loss: 1.398098
(Iteration 1391 / 2450) loss: 1.306147
(Iteration 1401 / 2450) loss: 1.411933
(Iteration 1411 / 2450) loss: 1.330717
(Iteration 1421 / 2450) loss: 1.488050
(Iteration 1431 / 2450) loss: 1.351308
(Iteration 1441 / 2450) loss: 1.301283
(Iteration 1451 / 2450) loss: 1.345304
(Iteration 1461 / 2450) loss: 1.413720
(Epoch 6 / 10) train acc: 0.529000; val_acc: 0.514000
(Iteration 1471 / 2450) loss: 1.198083
(Iteration 1481 / 2450) loss: 1.478806
(Iteration 1491 / 2450) loss: 1.376995
(Iteration 1501 / 2450) loss: 1.429425
(Iteration 1511 / 2450) loss: 1.377758
(Iteration 1521 / 2450) loss: 1.345896
(Iteration 1531 / 2450) loss: 1.249913
(Iteration 1541 / 2450) loss: 1.268721
(Iteration 1551 / 2450) loss: 1.343198
(Iteration 1561 / 2450) loss: 1.338622
(Iteration 1571 / 2450) loss: 1.312422
(Iteration 1581 / 2450) loss: 1.232372
(Iteration 1591 / 2450) loss: 1.467835
(Iteration 1601 / 2450) loss: 1.241809
(Iteration 1611 / 2450) loss: 1.372331
(Iteration 1621 / 2450) loss: 1.363680
(Iteration 1631 / 2450) loss: 1.380844
(Iteration 1641 / 2450) loss: 1.327067
(Iteration 1651 / 2450) loss: 1.292674
(Iteration 1661 / 2450) loss: 1.432018
(Iteration 1671 / 2450) loss: 1.306678
(Iteration 1681 / 2450) loss: 1.281083
(Iteration 1691 / 2450) loss: 1.299550
(Iteration 1701 / 2450) loss: 1.353815
(Iteration 1711 / 2450) loss: 1.447872
(Epoch 7 / 10) train acc: 0.542000; val_acc: 0.518000
(Iteration 1721 / 2450) loss: 1.373311
(Iteration 1731 / 2450) loss: 1.321872
```

```
(Iteration 1741 / 2450) loss: 1.424557
(Iteration 1751 / 2450) loss: 1.341724
(Iteration 1761 / 2450) loss: 1.248773
(Iteration 1771 / 2450) loss: 1.224022
(Iteration 1781 / 2450) loss: 1.245750
(Iteration 1791 / 2450) loss: 1.409639
(Iteration 1801 / 2450) loss: 1.381993
(Iteration 1811 / 2450) loss: 1.383563
(Iteration 1821 / 2450) loss: 1.292674
(Iteration 1831 / 2450) loss: 1.288933
(Iteration 1841 / 2450) loss: 1.496392
(Iteration 1851 / 2450) loss: 1.328678
(Iteration 1861 / 2450) loss: 1.446075
(Iteration 1871 / 2450) loss: 1.268318
(Iteration 1881 / 2450) loss: 1.498961
(Iteration 1891 / 2450) loss: 1.376395
(Iteration 1901 / 2450) loss: 1.392121
(Iteration 1911 / 2450) loss: 1.373025
(Iteration 1921 / 2450) loss: 1.227876
(Iteration 1931 / 2450) loss: 1.316135
(Iteration 1941 / 2450) loss: 1.197981
(Iteration 1951 / 2450) loss: 1.342773
(Epoch 8 / 10) train acc: 0.554000; val_acc: 0.512000
(Iteration 1961 / 2450) loss: 1.407067
(Iteration 1971 / 2450) loss: 1.356952
(Iteration 1981 / 2450) loss: 1.298525
(Iteration 1991 / 2450) loss: 1.328497
(Iteration 2001 / 2450) loss: 1.267785
(Iteration 2011 / 2450) loss: 1.349393
(Iteration 2021 / 2450) loss: 1.362988
(Iteration 2031 / 2450) loss: 1.269268
(Iteration 2041 / 2450) loss: 1.332639
(Iteration 2051 / 2450) loss: 1.326851
(Iteration 2061 / 2450) loss: 1.390439
(Iteration 2071 / 2450) loss: 1.298079
(Iteration 2081 / 2450) loss: 1.401963
(Iteration 2091 / 2450) loss: 1.246414
(Iteration 2101 / 2450) loss: 1.423599
(Iteration 2111 / 2450) loss: 1.355854
(Iteration 2121 / 2450) loss: 1.294452
(Iteration 2131 / 2450) loss: 1.338360
(Iteration 2141 / 2450) loss: 1.228762
(Iteration 2151 / 2450) loss: 1.322984
(Iteration 2161 / 2450) loss: 1.250192
(Iteration 2171 / 2450) loss: 1.420532
(Iteration 2181 / 2450) loss: 1.394822
(Iteration 2191 / 2450) loss: 1.361119
(Iteration 2201 / 2450) loss: 1.322465
(Epoch 9 / 10) train acc: 0.563000; val_acc: 0.526000
(Iteration 2211 / 2450) loss: 1.336937
(Iteration 2221 / 2450) loss: 1.429790
(Iteration 2231 / 2450) loss: 1.405557
(Iteration 2241 / 2450) loss: 1.086542
(Iteration 2251 / 2450) loss: 1.310014
(Iteration 2261 / 2450) loss: 1.252913
(Iteration 2271 / 2450) loss: 1.443422
(Iteration 2281 / 2450) loss: 1.207664
(Iteration 2291 / 2450) loss: 1.309647
(Iteration 2301 / 2450) loss: 1.204070
(Iteration 2311 / 2450) loss: 1.228339
(Iteration 2321 / 2450) loss: 1.356092
(Iteration 2331 / 2450) loss: 1.265508
(Iteration 2341 / 2450) loss: 1.370749
(Iteration 2351 / 2450) loss: 1.313774
(Iteration 2361 / 2450) loss: 1.275467
(Iteration 2371 / 2450) loss: 1.295278
(Iteration 2381 / 2450) loss: 1.209425
(Iteration 2391 / 2450) loss: 1.281593
(Iteration 2401 / 2450) loss: 1.216995
```

```
(Iteration 2411 / 2450) loss: 1.263418
(Iteration 2421 / 2450) loss: 1.199211
(Iteration 2431 / 2450) loss: 1.389708
(Iteration 2441 / 2450) loss: 1.254848
(Epoch 10 / 10) train acc: 0.576000; val_acc: 0.534000
(Iteration 1 / 2450) loss: 2.299404
(Epoch 0 / 10) train acc: 0.140000; val_acc: 0.146000
(Iteration 11 / 2450) loss: 2.228216
(Iteration 21 / 2450) loss: 2.152592
(Iteration 31 / 2450) loss: 2.080897
(Iteration 41 / 2450) loss: 1.969834
(Iteration 51 / 2450) loss: 1.936306
(Iteration 61 / 2450) loss: 1.922485
(Iteration 71 / 2450) loss: 1.909637
(Iteration 81 / 2450) loss: 1.784598
(Iteration 91 / 2450) loss: 1.818304
(Iteration 101 / 2450) loss: 1.771110
(Iteration 111 / 2450) loss: 1.773331
(Iteration 121 / 2450) loss: 1.740045
(Iteration 131 / 2450) loss: 1.807811
(Iteration 141 / 2450) loss: 1.830323
(Iteration 151 / 2450) loss: 1.733733
(Iteration 161 / 2450) loss: 1.774151
(Iteration 171 / 2450) loss: 1.716766
(Iteration 181 / 2450) loss: 1.636991
(Iteration 191 / 2450) loss: 1.687288
(Iteration 201 / 2450) loss: 1.665833
(Iteration 211 / 2450) loss: 1.697864
(Iteration 221 / 2450) loss: 1.703602
(Iteration 231 / 2450) loss: 1.563073
(Iteration 241 / 2450) loss: 1.568409
(Epoch 1 / 10) train acc: 0.428000; val_acc: 0.435000
(Iteration 251 / 2450) loss: 1.625733
(Iteration 261 / 2450) loss: 1.637794
(Iteration 271 / 2450) loss: 1.634065
(Iteration 281 / 2450) loss: 1.748086
(Iteration 291 / 2450) loss: 1.617113
(Iteration 301 / 2450) loss: 1.614901
(Iteration 311 / 2450) loss: 1.638776
(Iteration 321 / 2450) loss: 1.559714
(Iteration 331 / 2450) loss: 1.575609
(Iteration 341 / 2450) loss: 1.480461
(Iteration 351 / 2450) loss: 1.574916
(Iteration 361 / 2450) loss: 1.553246
(Iteration 371 / 2450) loss: 1.504287
(Iteration 381 / 2450) loss: 1.499402
(Iteration 391 / 2450) loss: 1.540788
(Iteration 401 / 2450) loss: 1.562162
(Iteration 411 / 2450) loss: 1.498230
(Iteration 421 / 2450) loss: 1.531758
(Iteration 431 / 2450) loss: 1.494927
(Iteration 441 / 2450) loss: 1.570338
(Iteration 451 / 2450) loss: 1.603768
(Iteration 461 / 2450) loss: 1.604422
(Iteration 471 / 2450) loss: 1.492591
(Iteration 481 / 2450) loss: 1.490097
(Epoch 2 / 10) train acc: 0.469000; val_acc: 0.465000
(Iteration 491 / 2450) loss: 1.418956
(Iteration 501 / 2450) loss: 1.542555
(Iteration 511 / 2450) loss: 1.412967
(Iteration 521 / 2450) loss: 1.571352
(Iteration 531 / 2450) loss: 1.609651
(Iteration 541 / 2450) loss: 1.574309
(Iteration 551 / 2450) loss: 1.541248
(Iteration 561 / 2450) loss: 1.594207
(Iteration 571 / 2450) loss: 1.404603
(Iteration 581 / 2450) loss: 1.480139
(Iteration 591 / 2450) loss: 1.538158
(Iteration 601 / 2450) loss: 1.404489
```

```
(Iteration 611 / 2450) loss: 1.669519
(Iteration 621 / 2450) loss: 1.324889
(Iteration 631 / 2450) loss: 1.473916
(Iteration 641 / 2450) loss: 1.476052
(Iteration 651 / 2450) loss: 1.425997
(Iteration 661 / 2450) loss: 1.501530
(Iteration 671 / 2450) loss: 1.399951
(Iteration 681 / 2450) loss: 1.401339
(Iteration 691 / 2450) loss: 1.534271
(Iteration 701 / 2450) loss: 1.558348
(Iteration 711 / 2450) loss: 1.586969
(Iteration 721 / 2450) loss: 1.378051
(Iteration 731 / 2450) loss: 1.472686
(Epoch 3 / 10) train acc: 0.488000; val_acc: 0.476000
(Iteration 741 / 2450) loss: 1.354568
(Iteration 751 / 2450) loss: 1.301806
(Iteration 761 / 2450) loss: 1.513828
(Iteration 771 / 2450) loss: 1.452991
(Iteration 781 / 2450) loss: 1.490318
(Iteration 791 / 2450) loss: 1.449668
(Iteration 801 / 2450) loss: 1.570065
(Iteration 811 / 2450) loss: 1.455577
(Iteration 821 / 2450) loss: 1.489975
(Iteration 831 / 2450) loss: 1.386252
(Iteration 841 / 2450) loss: 1.390982
(Iteration 851 / 2450) loss: 1.328363
(Iteration 861 / 2450) loss: 1.415266
(Iteration 871 / 2450) loss: 1.346509
(Iteration 881 / 2450) loss: 1.404249
(Iteration 891 / 2450) loss: 1.351382
(Iteration 901 / 2450) loss: 1.533100
(Iteration 911 / 2450) loss: 1.505557
(Iteration 921 / 2450) loss: 1.377168
(Iteration 931 / 2450) loss: 1.248938
(Iteration 941 / 2450) loss: 1.454387
(Iteration 951 / 2450) loss: 1.439001
(Iteration 961 / 2450) loss: 1.508369
(Iteration 971 / 2450) loss: 1.404391
(Epoch 4 / 10) train acc: 0.503000; val_acc: 0.477000
(Iteration 981 / 2450) loss: 1.383898
(Iteration 991 / 2450) loss: 1.426024
(Iteration 1001 / 2450) loss: 1.380700
(Iteration 1011 / 2450) loss: 1.402513
(Iteration 1021 / 2450) loss: 1.302179
(Iteration 1031 / 2450) loss: 1.268368
(Iteration 1041 / 2450) loss: 1.375602
(Iteration 1051 / 2450) loss: 1.329166
(Iteration 1061 / 2450) loss: 1.358613
(Iteration 1071 / 2450) loss: 1.459635
(Iteration 1081 / 2450) loss: 1.530684
(Iteration 1091 / 2450) loss: 1.407698
(Iteration 1101 / 2450) loss: 1.218712
(Iteration 1111 / 2450) loss: 1.220622
(Iteration 1121 / 2450) loss: 1.354323
(Iteration 1131 / 2450) loss: 1.299667
(Iteration 1141 / 2450) loss: 1.444227
(Iteration 1151 / 2450) loss: 1.340216
(Iteration 1161 / 2450) loss: 1.494445
(Iteration 1171 / 2450) loss: 1.420634
(Iteration 1181 / 2450) loss: 1.246824
(Iteration 1191 / 2450) loss: 1.411436
(Iteration 1201 / 2450) loss: 1.326497
(Iteration 1211 / 2450) loss: 1.353450
(Iteration 1221 / 2450) loss: 1.269083
(Epoch 5 / 10) train acc: 0.559000; val_acc: 0.487000
(Iteration 1231 / 2450) loss: 1.235708
(Iteration 1241 / 2450) loss: 1.351695
(Iteration 1251 / 2450) loss: 1.354830
(Iteration 1261 / 2450) loss: 1.372343
```

```
(Iteration 1271 / 2450) loss: 1.339122
(Iteration 1281 / 2450) loss: 1.385760
(Iteration 1291 / 2450) loss: 1.246537
(Iteration 1301 / 2450) loss: 1.306262
(Iteration 1311 / 2450) loss: 1.406660
(Iteration 1321 / 2450) loss: 1.458700
(Iteration 1331 / 2450) loss: 1.345401
(Iteration 1341 / 2450) loss: 1.268546
(Iteration 1351 / 2450) loss: 1.353159
(Iteration 1361 / 2450) loss: 1.431117
(Iteration 1371 / 2450) loss: 1.407764
(Iteration 1381 / 2450) loss: 1.428515
(Iteration 1391 / 2450) loss: 1.401980
(Iteration 1401 / 2450) loss: 1.353785
(Iteration 1411 / 2450) loss: 1.173027
(Iteration 1421 / 2450) loss: 1.388208
(Iteration 1431 / 2450) loss: 1.386620
(Iteration 1441 / 2450) loss: 1.300103
(Iteration 1451 / 2450) loss: 1.409683
(Iteration 1461 / 2450) loss: 1.195710
(Epoch 6 / 10) train acc: 0.576000; val_acc: 0.481000
(Iteration 1471 / 2450) loss: 1.216651
(Iteration 1481 / 2450) loss: 1.167857
(Iteration 1491 / 2450) loss: 1.245672
(Iteration 1501 / 2450) loss: 1.408744
(Iteration 1511 / 2450) loss: 1.362521
(Iteration 1521 / 2450) loss: 1.224297
(Iteration 1531 / 2450) loss: 1.226797
(Iteration 1541 / 2450) loss: 1.263582
(Iteration 1551 / 2450) loss: 1.454857
(Iteration 1561 / 2450) loss: 1.359151
(Iteration 1571 / 2450) loss: 1.483940
(Iteration 1581 / 2450) loss: 1.411976
(Iteration 1591 / 2450) loss: 1.297835
(Iteration 1601 / 2450) loss: 1.363584
(Iteration 1611 / 2450) loss: 1.269396
(Iteration 1621 / 2450) loss: 1.167956
(Iteration 1631 / 2450) loss: 1.302924
(Iteration 1641 / 2450) loss: 1.320946
(Iteration 1651 / 2450) loss: 1.303204
(Iteration 1661 / 2450) loss: 1.369169
(Iteration 1671 / 2450) loss: 1.401713
(Iteration 1681 / 2450) loss: 1.319241
(Iteration 1691 / 2450) loss: 1.317585
(Iteration 1701 / 2450) loss: 1.417822
(Iteration 1711 / 2450) loss: 1.378067
(Epoch 7 / 10) train acc: 0.562000; val_acc: 0.497000
(Iteration 1721 / 2450) loss: 1.396727
(Iteration 1731 / 2450) loss: 1.425352
(Iteration 1741 / 2450) loss: 1.253821
(Iteration 1751 / 2450) loss: 1.384034
(Iteration 1761 / 2450) loss: 1.350305
(Iteration 1771 / 2450) loss: 1.291718
(Iteration 1781 / 2450) loss: 1.264170
(Iteration 1791 / 2450) loss: 1.204820
(Iteration 1801 / 2450) loss: 1.295378
(Iteration 1811 / 2450) loss: 1.247565
(Iteration 1821 / 2450) loss: 1.161550
(Iteration 1831 / 2450) loss: 1.282808
(Iteration 1841 / 2450) loss: 1.291747
(Iteration 1851 / 2450) loss: 1.226963
(Iteration 1861 / 2450) loss: 1.168692
(Iteration 1871 / 2450) loss: 1.223844
(Iteration 1881 / 2450) loss: 1.364989
(Iteration 1891 / 2450) loss: 1.257686
(Iteration 1901 / 2450) loss: 1.166312
(Iteration 1911 / 2450) loss: 1.204874
(Iteration 1921 / 2450) loss: 1.266848
(Iteration 1931 / 2450) loss: 1.391833
```

```
(Iteration 1941 / 2450) loss: 1.320299
(Iteration 1951 / 2450) loss: 1.230980
(Epoch 8 / 10) train acc: 0.562000; val_acc: 0.494000
(Iteration 1961 / 2450) loss: 1.317515
(Iteration 1971 / 2450) loss: 1.239543
(Iteration 1981 / 2450) loss: 1.126723
(Iteration 1991 / 2450) loss: 1.438992
(Iteration 2001 / 2450) loss: 1.203190
(Iteration 2011 / 2450) loss: 1.330240
(Iteration 2021 / 2450) loss: 1.378941
(Iteration 2031 / 2450) loss: 1.166048
(Iteration 2041 / 2450) loss: 1.260948
(Iteration 2051 / 2450) loss: 1.311082
(Iteration 2061 / 2450) loss: 1.100234
(Iteration 2071 / 2450) loss: 1.204254
(Iteration 2081 / 2450) loss: 1.200223
(Iteration 2091 / 2450) loss: 1.335809
(Iteration 2101 / 2450) loss: 1.257428
(Iteration 2111 / 2450) loss: 1.317562
(Iteration 2121 / 2450) loss: 1.125827
(Iteration 2131 / 2450) loss: 1.153936
(Iteration 2141 / 2450) loss: 1.295766
(Iteration 2151 / 2450) loss: 1.321448
(Iteration 2161 / 2450) loss: 1.148166
(Iteration 2171 / 2450) loss: 1.162744
(Iteration 2181 / 2450) loss: 1.245360
(Iteration 2191 / 2450) loss: 1.318681
(Iteration 2201 / 2450) loss: 1.196430
(Epoch 9 / 10) train acc: 0.570000; val_acc: 0.495000
(Iteration 2211 / 2450) loss: 1.375882
(Iteration 2221 / 2450) loss: 1.333221
(Iteration 2231 / 2450) loss: 1.355763
(Iteration 2241 / 2450) loss: 1.136080
(Iteration 2251 / 2450) loss: 1.182891
(Iteration 2261 / 2450) loss: 1.188033
(Iteration 2271 / 2450) loss: 1.211487
(Iteration 2281 / 2450) loss: 1.387566
(Iteration 2291 / 2450) loss: 1.256158
(Iteration 2301 / 2450) loss: 1.276345
(Iteration 2311 / 2450) loss: 1.260205
(Iteration 2321 / 2450) loss: 1.248020
(Iteration 2331 / 2450) loss: 1.154385
(Iteration 2341 / 2450) loss: 1.240530
(Iteration 2351 / 2450) loss: 1.211996
(Iteration 2361 / 2450) loss: 1.198929
(Iteration 2371 / 2450) loss: 1.209735
(Iteration 2381 / 2450) loss: 1.195583
(Iteration 2391 / 2450) loss: 1.309717
(Iteration 2401 / 2450) loss: 1.255993
(Iteration 2411 / 2450) loss: 1.197183
(Iteration 2421 / 2450) loss: 1.193336
(Iteration 2431 / 2450) loss: 1.271226
(Iteration 2441 / 2450) loss: 1.304555
(Epoch 10 / 10) train acc: 0.596000; val_acc: 0.508000
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

In [47]:

```
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
Validation set accuracy:  0.508
```

In [48]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Test set accuracy:  0.509
```

Inline Question 2:

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

$Your Answer$: 1, 3

$Your Explanation$: 일반적으로, train 셋의 데이터를 늘리거나 regularization strength를 높이면 test셋에서의 예측값을 train값과 더 비슷하게 만들 수 있다.

2번과 같은 경우는, hidden units를 높이면 오버피팅이 될 가능성도 있으므로, 적절한 값은 하이퍼파라미터 튜닝을 통해 결정된다.