In [ ]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'assignment4'
FOLDERNAME = 'CS6353/Assignments/assignment4/assignment4/'
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 /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
mount("/content/drive", force_remount=True).
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4/cs6353/datasets
--2024-11-03 05:03:49--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connecte
d.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  31.2MB/s    in 5.9s

2024-11-03 05:03:55 (27.7 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/17049807
1]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4
```

Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer

and modify your fully-connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

In [ ]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.fc_net import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_gradient_
from cs6353.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))))

# You can ignore the message that asks you to run Python script for now.
# It will be required in the second part of the assignment.
```

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

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

data = get_CIFAR10_data()
for k, v in 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,)
```

Dropout forward pass

In the file `cs6353/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

In [ ]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `cs6353/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

In [ ]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param
```

```
# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

Answer:

If we don't divide the values by p during training with dropout, we're using vanilla dropout instead of inverted dropout. This means at test time, we would need to scale the outputs by p to ensure the activations match the expected values from training.

Without scaling, the test-time activations would be too large since, during training, neurons see only a fraction of their inputs due to dropout, while at test time, all neurons are active. This discrepancy could lead to unstable predictions and exploding gradients.

Inverted dropout solves this by scaling the activations during training (multiplying by 1 / p). This keeps the expected activations consistent between training and testing, eliminating the need for test-time scaling and simplifying the deployment process.

Inverted dropout is preferred because it ensures consistent behavior and avoids the need for adjustments during inference.

Fully-connected nets with Dropout

In the file `cs6353/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

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

for dropout in [1, 0.75, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

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

```
  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 1.69e-10

Running check with dropout =  0.75
Initial loss:  2.3028487331533216
W1 relative error: 7.87e-07
W2 relative error: 1.12e-07
W3 relative error: 1.47e-07
b1 relative error: 1.83e-08
b2 relative error: 4.32e-09
b3 relative error: 1.75e-10

Running check with dropout =  0.5
Initial loss:  2.30427592207859
W1 relative error: 3.11e-07
W2 relative error: 2.48e-08
W3 relative error: 6.43e-08
b1 relative error: 5.37e-09
b2 relative error: 1.91e-09
b3 relative error: 1.85e-10
```

Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

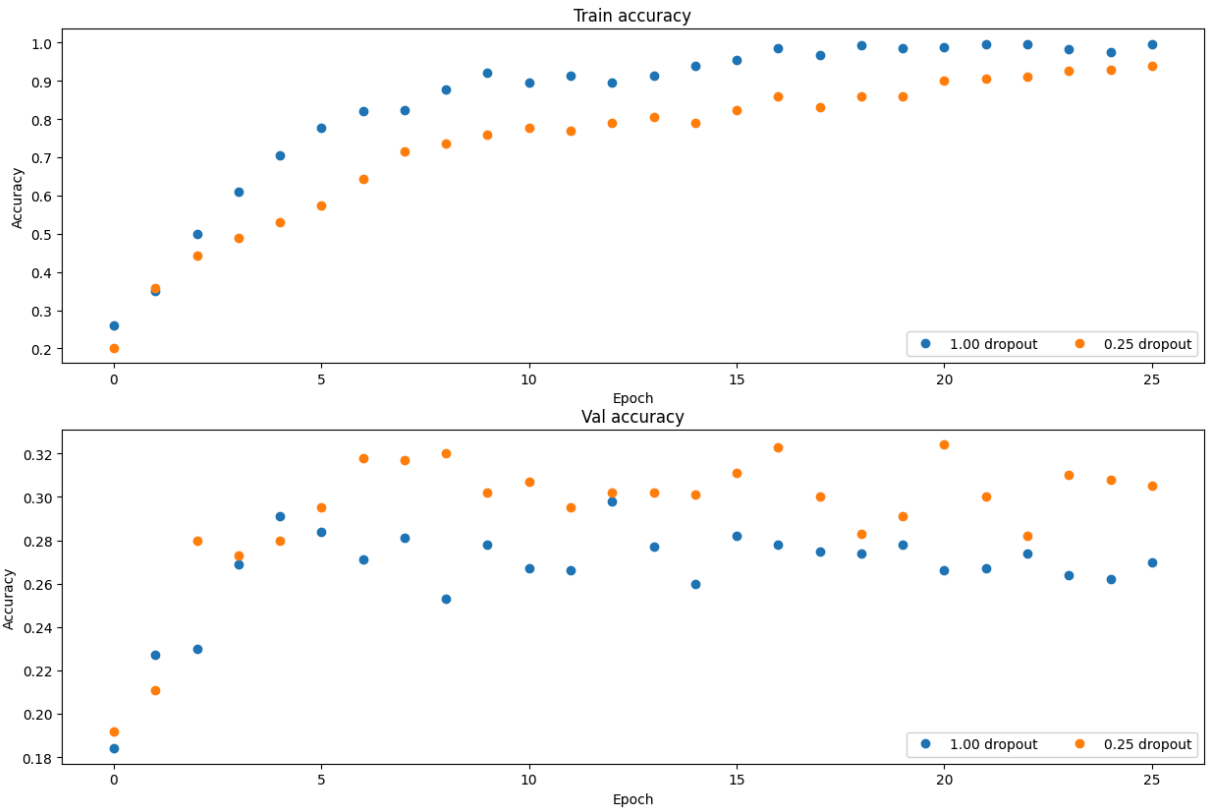
In [ ]:
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}
```

```
solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.350000; val_acc: 0.227000
(Epoch 2 / 25) train acc: 0.500000; val_acc: 0.230000
(Epoch 3 / 25) train acc: 0.610000; val_acc: 0.269000
(Epoch 4 / 25) train acc: 0.704000; val_acc: 0.291000
(Epoch 5 / 25) train acc: 0.778000; val_acc: 0.284000
(Epoch 6 / 25) train acc: 0.820000; val_acc: 0.271000
(Epoch 7 / 25) train acc: 0.824000; val_acc: 0.281000
(Epoch 8 / 25) train acc: 0.876000; val_acc: 0.253000
(Epoch 9 / 25) train acc: 0.920000; val_acc: 0.278000
(Epoch 10 / 25) train acc: 0.896000; val_acc: 0.267000
(Epoch 11 / 25) train acc: 0.912000; val_acc: 0.266000
(Epoch 12 / 25) train acc: 0.894000; val_acc: 0.298000
(Epoch 13 / 25) train acc: 0.914000; val_acc: 0.277000
(Epoch 14 / 25) train acc: 0.938000; val_acc: 0.260000
(Epoch 15 / 25) train acc: 0.954000; val_acc: 0.282000
(Epoch 16 / 25) train acc: 0.984000; val_acc: 0.278000
(Epoch 17 / 25) train acc: 0.966000; val_acc: 0.275000
(Epoch 18 / 25) train acc: 0.992000; val_acc: 0.274000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.278000
(Epoch 20 / 25) train acc: 0.988000; val_acc: 0.266000
(Iteration 101 / 125) loss: 0.068826
(Epoch 21 / 25) train acc: 0.996000; val_acc: 0.267000
(Epoch 22 / 25) train acc: 0.996000; val_acc: 0.274000
(Epoch 23 / 25) train acc: 0.982000; val_acc: 0.264000
(Epoch 24 / 25) train acc: 0.974000; val_acc: 0.262000
(Epoch 25 / 25) train acc: 0.996000; val_acc: 0.270000
0.25
(Iteration 1 / 125) loss: 17.045781
(Epoch 0 / 25) train acc: 0.202000; val_acc: 0.192000
(Epoch 1 / 25) train acc: 0.358000; val_acc: 0.211000
(Epoch 2 / 25) train acc: 0.444000; val_acc: 0.280000
(Epoch 3 / 25) train acc: 0.488000; val_acc: 0.273000
(Epoch 4 / 25) train acc: 0.530000; val_acc: 0.280000
(Epoch 5 / 25) train acc: 0.574000; val_acc: 0.295000
(Epoch 6 / 25) train acc: 0.644000; val_acc: 0.318000
(Epoch 7 / 25) train acc: 0.714000; val_acc: 0.317000
(Epoch 8 / 25) train acc: 0.736000; val_acc: 0.320000
(Epoch 9 / 25) train acc: 0.760000; val_acc: 0.302000
(Epoch 10 / 25) train acc: 0.776000; val_acc: 0.307000
(Epoch 11 / 25) train acc: 0.770000; val_acc: 0.295000
(Epoch 12 / 25) train acc: 0.790000; val_acc: 0.302000
(Epoch 13 / 25) train acc: 0.804000; val_acc: 0.302000
(Epoch 14 / 25) train acc: 0.790000; val_acc: 0.301000
(Epoch 15 / 25) train acc: 0.824000; val_acc: 0.311000
(Epoch 16 / 25) train acc: 0.860000; val_acc: 0.323000
(Epoch 17 / 25) train acc: 0.832000; val_acc: 0.300000
(Epoch 18 / 25) train acc: 0.858000; val_acc: 0.283000
(Epoch 19 / 25) train acc: 0.858000; val_acc: 0.291000
(Epoch 20 / 25) train acc: 0.900000; val_acc: 0.324000
(Iteration 101 / 125) loss: 5.281125
(Epoch 21 / 25) train acc: 0.906000; val_acc: 0.300000
(Epoch 22 / 25) train acc: 0.910000; val_acc: 0.282000
(Epoch 23 / 25) train acc: 0.926000; val_acc: 0.310000
```

```
(Epoch 24 / 25) train acc: 0.928000; val_acc: 0.308000
(Epoch 25 / 25) train acc: 0.938000; val_acc: 0.305000
```

In [ ]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

When comparing training and validation accuracies with and without dropout, the results highlight the role of dropout as a regularizer, particularly in reducing overfitting.

Without Dropout: The model shows clear signs of overfitting. Training accuracy is extremely high (99.6% at epoch 25), while validation accuracy is much lower (27% at epoch 25). This large gap indicates that the model is fitting the training data too well but struggling to generalize to new, unseen data.

With Dropout: Dropout reduces the training accuracy slightly (93.8% at epoch 25), but the validation accuracy improves (30.5% at epoch 25). This suggests that dropout regularizes the model, preventing it from overfitting and helping it generalize better to the validation set.

Conclusion:

Dropout effectively narrows the gap between training and validation performance. While it may reduce training accuracy slightly, it leads to better validation accuracy, indicating improved generalization. This supports the conclusion that dropout helps control model complexity, reducing overfitting and improving performance on unseen data.

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

If you reduce the size of the hidden layers in a fully-connected network with dropout (parameterized by keep probability p), you don't need to adjust p for overfitting concerns. Dropout adjusts proportionally, meaning the number of neurons dropped is scaled according to the layer's size.

For example, let's say a hidden layer has n = 512 neurons and you use n = 512 and p = 0.25. Then number of dropped neurons would be p * n = 128, leaving 384 neurons active. Now, if

you reduce the hidden layer size to 256 neurons and keep p = 0.25. Then number of dropped neurons would be p * n = 64, leaving 192 neurons active.

Since dropout scales the number of dropped neurons proportionally with the layer size, there's no need to modify p when decreasing the number of neurons in the hidden layers.

In [2]:
```
# Uncomment this block if you are using colabVM

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

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'assignment4'
FOLDERNAME = 'CS6353/Assignments/assignment4/assignment4/'
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 /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4/cs6353/datasets
--2024-11-04 14:52:34--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connecte
d.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  42.3MB/s    in 4.0s

2024-11-04 14:52:38 (40.8 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/17049807
1]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/CS6353/Assignments/assignment4/assignment4
```

Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

In [3]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs6353.classifiers.cnn import *
from cs6353.data_utils import get_CIFAR10_data
from cs6353.gradient_check import eval_numerical_gradient_array, eval_numerical_gra
from cs6353.layers import *
from cs6353.fast_layers import *
from cs6353.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))))

# You can ignore the message that asks you to run Python script for now.
# It will be required in the later part of the assignment.
```

In [4]:
```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in 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,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs6353/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

In [5]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around 1e-8
print ('Testing conv_forward_naive')
print ('difference: ', rel_error(out, correct_out))
```

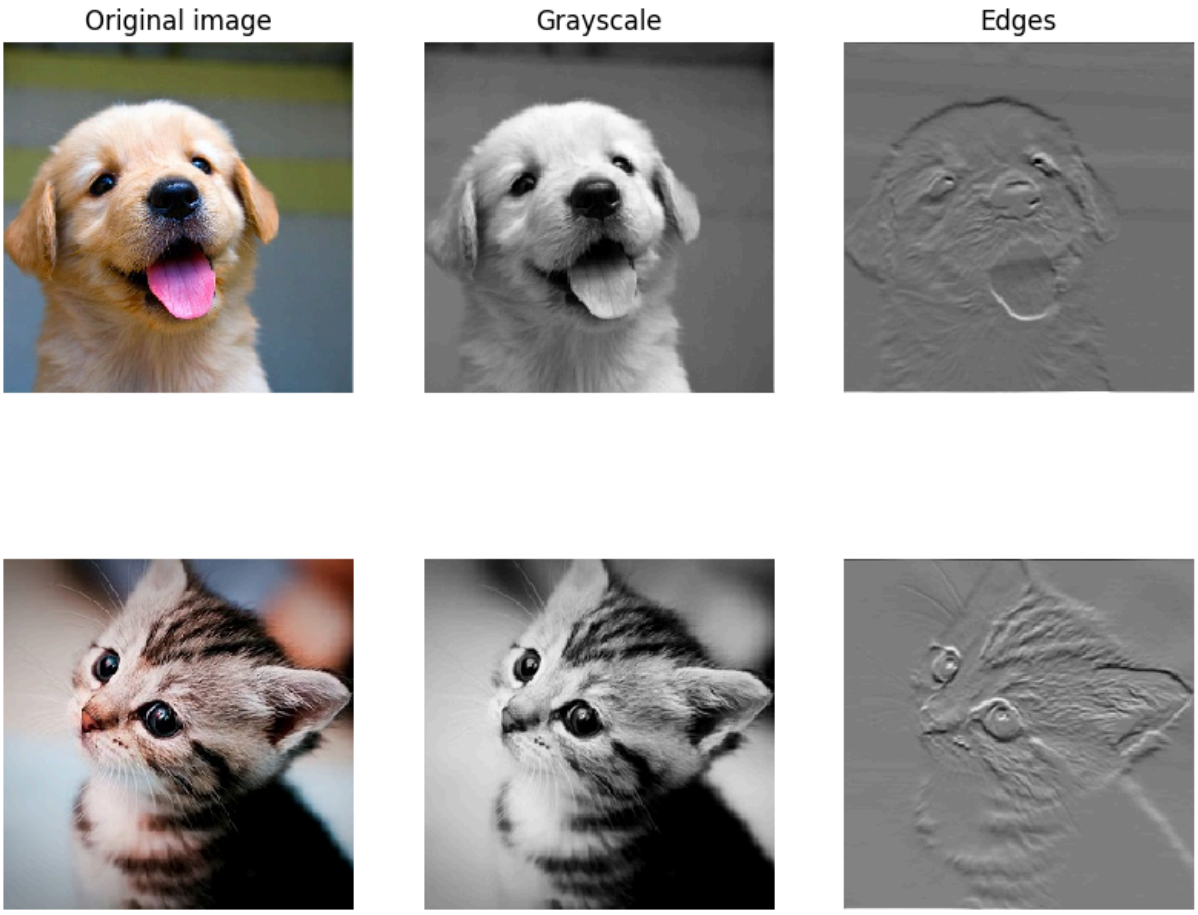
```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

In [6]:
```
import imageio.v2 as imageio
from PIL import Image

kitten, puppy = imageio.imread('kitten.jpg'), imageio.imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
```

```
kitten_cropped = kitten[:, int(d/2):int(-d/2), :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = np.array(Image.fromarray(puppy).resize((img_size, img_size))).trans
x[1, :, :, :] = np.array(Image.fromarray(kitten_cropped).resize((img_size, img_size

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```



Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs6353/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

In [7]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_p
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_p
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_p

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around 1e-9'
print ('Testing conv_backward_naive function')
```

```
print ('dx error: ', rel_error(dx, dx_num))
print ('dw error: ', rel_error(dw, dw_num))
print ('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  2.5072910768288887e-09
dw error:  3.4057977751826577e-10
db error:  8.058625830336249e-12
```

Max pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs6353/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

In [8]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be around 1e-8.
print ('Testing max_pool_forward_naive function:')
print ('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

Max pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs6353/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

In [9]:
```
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_par

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be around 1e-12
print ('Testing max_pool_backward_naive function:')
print ('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.275625745923618e-12
```

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs6353/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs6353` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

In [10]:
```
!pip install colab-xterm
%load_ext colabxterm
%xterm

# Once the terminal loads, run the following commands
# cd cs6353
# python setup.py build_ext --inplace
```

```
Collecting colab-xterm
  Downloading colab_xterm-0.2.0-py3-none-any.whl.metadata (1.2 kB)
Requirement already satisfied: ptyprocess~=0.7.0 in /usr/local/lib/python3.10/dist-p
ackages (from colab-xterm) (0.7.0)
Requirement already satisfied: tornado>5.1 in /usr/local/lib/python3.10/dist-package
s (from colab-xterm) (6.3.3)
Downloading colab_xterm-0.2.0-py3-none-any.whl (115 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.6/115.6 kB 2.2 MB/s eta 0:00:00
Installing collected packages: colab-xterm
Successfully installed colab-xterm-0.2.0
Launching Xterm...
```

In [11]:
```
from cs6353.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print ('Testing conv_forward_fast:')
print ('Naive: %fs' % (t1 - t0))
print ('Fast: %fs' % (t2 - t1))
print ('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print ('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print ()
print ('Testing conv_backward_fast:')
print ('Naive: %fs' % (t1 - t0))
print ('Fast: %fs' % (t2 - t1))
print ('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print ('dx difference: ', rel_error(dx_naive, dx_fast))
print ('dw difference: ', rel_error(dw_naive, dw_fast))
print ('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 7.254762s
Fast: 0.016461s
Speedup: 440.727050x
Difference:  8.242836248261497e-11

Testing conv_backward_fast:
Naive: 8.643726s
Fast: 0.012475s
Speedup: 692.883094x
dx difference:  3.3973352731104886e-11
dw difference:  1.4012561808818972e-12
db difference:  0.0
```

In [12]:
```
from cs6353.fast_layers import max_pool_forward_fast, max_pool_backward_fast

x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print ('Testing pool_forward_fast:')
print ('Naive: %fs' % (t1 - t0))
print ('fast: %fs' % (t2 - t1))
print ('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print ('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print ()
print ('Testing pool_backward_fast:')
print ('Naive: %fs' % (t1 - t0))
print ('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print ('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.146966s
fast: 0.005592s
speedup: 26.281006x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.362660s
speedup: 23.947668x
dx difference:  0.0
```

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs6353/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

In [14]:
```
from cs6353.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, co
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, co
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, co

print ('Testing conv_relu_pool')
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 conv_relu_pool
dx error:  3.431316328713469e-08
dw error:  1.546113609990029e-09
db error:  2.3424001937131512e-11
```

In [15]:
```
from cs6353.layer_utils import conv_relu_forward, conv_relu_backward

x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_pa
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_pa
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_pa

print ('Testing conv_relu:')
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 conv_relu:
dx error:  4.401535025725687e-09
dw error:  3.8532111469111795e-09
db error:  1.1245410184440985e-11
```

Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs6353/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Run the following cells to help you debug:

Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

In [16]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print ('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print ('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.3025822530162663
Initial loss (with regularization):  2.5086330233444136
```

Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

In [17]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)
```

```
model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=F
    e = rel_error(param_grad_num, grads[param_name])
    print ('%s max relative error: %e' % (param_name, rel_error(param_grad_num, gra
```

```
W1 max relative error: 6.584815e-04
W2 max relative error: 9.681594e-03
W3 max relative error: 4.585888e-05
b1 max relative error: 9.519797e-06
b2 max relative error: 2.136966e-07
b3 max relative error: 1.140520e-09
```

Overfit small data

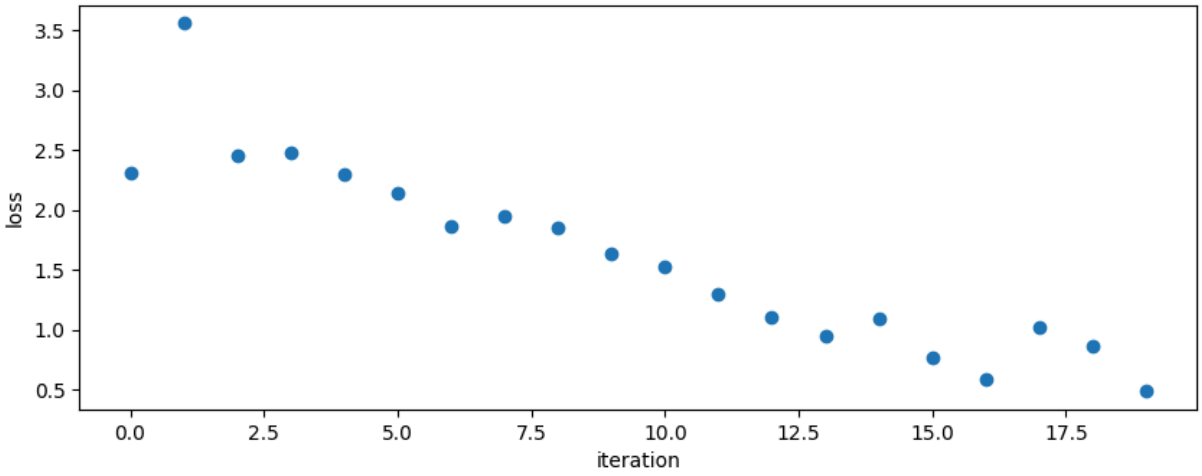
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

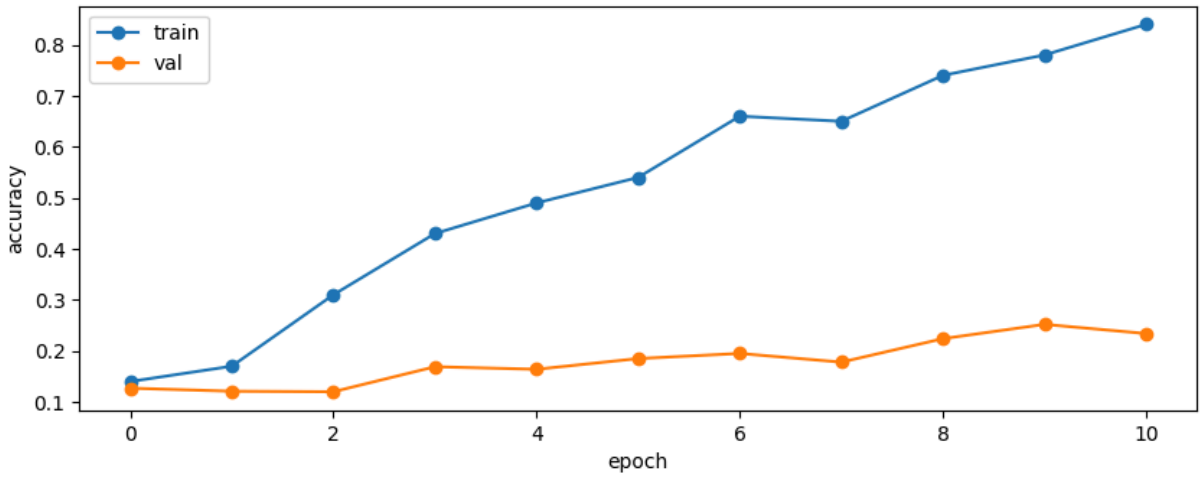
In [18]:
```
num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 20) loss: 2.314161
(Epoch 0 / 10) train acc: 0.140000; val_acc: 0.127000
(Iteration 2 / 20) loss: 3.556790
(Epoch 1 / 10) train acc: 0.170000; val_acc: 0.121000
(Iteration 3 / 20) loss: 2.449610
(Iteration 4 / 20) loss: 2.474680
(Epoch 2 / 10) train acc: 0.310000; val_acc: 0.120000
(Iteration 5 / 20) loss: 2.297068
(Iteration 6 / 20) loss: 2.141123
(Epoch 3 / 10) train acc: 0.430000; val_acc: 0.169000
(Iteration 7 / 20) loss: 1.864641
(Iteration 8 / 20) loss: 1.949257
(Epoch 4 / 10) train acc: 0.490000; val_acc: 0.164000
(Iteration 9 / 20) loss: 1.845802
(Iteration 10 / 20) loss: 1.631720
(Epoch 5 / 10) train acc: 0.540000; val_acc: 0.185000
(Iteration 11 / 20) loss: 1.525881
(Iteration 12 / 20) loss: 1.298279
(Epoch 6 / 10) train acc: 0.660000; val_acc: 0.195000
(Iteration 13 / 20) loss: 1.106351
(Iteration 14 / 20) loss: 0.940572
(Epoch 7 / 10) train acc: 0.650000; val_acc: 0.178000
(Iteration 15 / 20) loss: 1.087315
(Iteration 16 / 20) loss: 0.760858
(Epoch 8 / 10) train acc: 0.740000; val_acc: 0.224000
(Iteration 17 / 20) loss: 0.584596
(Iteration 18 / 20) loss: 1.019198
(Epoch 9 / 10) train acc: 0.780000; val_acc: 0.252000
(Iteration 19 / 20) loss: 0.860663
(Iteration 20 / 20) loss: 0.487877
(Epoch 10 / 10) train acc: 0.840000; val_acc: 0.234000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

In [19]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```





Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

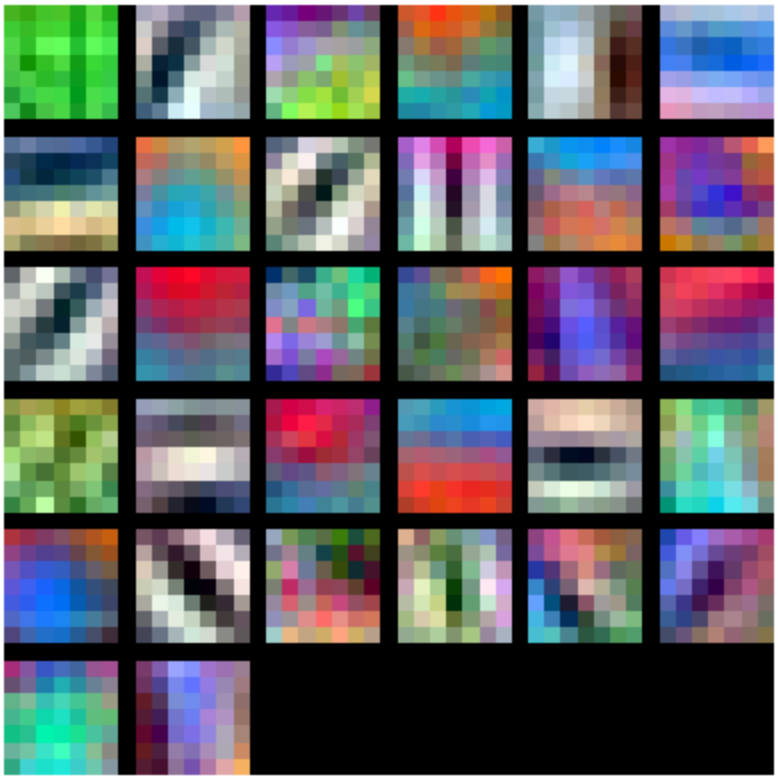
In [20]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304399
(Epoch 0 / 1) train acc: 0.098000; val_acc: 0.119000
(Iteration 21 / 980) loss: 2.202433
(Iteration 41 / 980) loss: 2.212356
(Iteration 61 / 980) loss: 2.045424
(Iteration 81 / 980) loss: 2.247800
(Iteration 101 / 980) loss: 1.935511
(Iteration 121 / 980) loss: 1.711983
(Iteration 141 / 980) loss: 1.880394
(Iteration 161 / 980) loss: 1.563806
(Iteration 181 / 980) loss: 1.820792
(Iteration 201 / 980) loss: 1.572483
(Iteration 221 / 980) loss: 1.674796
(Iteration 241 / 980) loss: 1.606313
(Iteration 261 / 980) loss: 1.911399
(Iteration 281 / 980) loss: 1.708723
(Iteration 301 / 980) loss: 2.144878
(Iteration 321 / 980) loss: 1.811579
(Iteration 341 / 980) loss: 1.662014
(Iteration 361 / 980) loss: 1.664599
(Iteration 381 / 980) loss: 1.828386
(Iteration 401 / 980) loss: 1.551436
(Iteration 421 / 980) loss: 1.327184
(Iteration 441 / 980) loss: 1.581322
(Iteration 461 / 980) loss: 1.653154
(Iteration 481 / 980) loss: 1.663376
(Iteration 501 / 980) loss: 1.783372
(Iteration 521 / 980) loss: 1.998902
(Iteration 541 / 980) loss: 1.396503
(Iteration 561 / 980) loss: 1.417825
(Iteration 581 / 980) loss: 1.384447
(Iteration 601 / 980) loss: 1.475926
(Iteration 621 / 980) loss: 1.603633
(Iteration 641 / 980) loss: 1.426967
(Iteration 661 / 980) loss: 1.581402
(Iteration 681 / 980) loss: 1.626013
(Iteration 701 / 980) loss: 1.531781
(Iteration 721 / 980) loss: 1.983411
(Iteration 741 / 980) loss: 1.412164
(Iteration 761 / 980) loss: 1.194317
(Iteration 781 / 980) loss: 1.731663
(Iteration 801 / 980) loss: 1.657997
(Iteration 821 / 980) loss: 1.415339
(Iteration 841 / 980) loss: 1.456162
(Iteration 861 / 980) loss: 1.704642
(Iteration 881 / 980) loss: 1.476581
(Iteration 901 / 980) loss: 1.730761
(Iteration 921 / 980) loss: 1.437530
(Iteration 941 / 980) loss: 1.363330
(Iteration 961 / 980) loss: 1.142858
(Epoch 1 / 1) train acc: 0.459000; val_acc: 0.474000
```

Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

In [22]:
```
from cs6353.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. Batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different imagesand different

locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

Spatial batch normalization: forward

In the file `cs6353/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

In [23]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print ('Before spatial batch normalization:')
print ('  Shape: ', x.shape)
print ('  Means: ', x.mean(axis=(0, 2, 3)))
print ('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print ('After spatial batch normalization:')
print ('  Shape: ', out.shape)
print ('  Means: ', out.mean(axis=(0, 2, 3)))
print ('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print ('After spatial batch normalization (nontrivial gamma, beta):')
print ('  Shape: ', out.shape)
print ('  Means: ', out.mean(axis=(0, 2, 3)))
print ('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [10.94581435  9.98849596  9.65853988]
  Stds:  [4.39531353 3.31603813 4.0412494 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [-5.55111512e-16 -3.49720253e-16  3.20576898e-16]
  Stds:  [0.99999974 0.99999955 0.99999969]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999922 3.99999818 4.99999847]
```

In [24]:
```
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
```

```
# checking the means and variances of activations after a test-time
# forward pass.

N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print ('After spatial batch normalization (test-time):')
print ('  means: ', a_norm.mean(axis=(0, 2, 3)))
print ('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [0.05533644 0.03200853 0.04807407 0.02680903]
  stds:  [1.03479481 1.00978012 0.94467793 1.02187916]
```

Spatial batch normalization: backward

In the file `cs6353/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

In [25]:
```
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print ('dx error: ', rel_error(dx_num, dx))
print ('dgamma error: ', rel_error(da_num, dgamma))
print ('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  2.8947422733425884e-08
dgamma error:  3.778378178946145e-12
dbeta error:  6.283818464040015e-12
```

Experiment!

Experiment and try to get the best performance that you can on CIFAR-10 using a ConvNet. Here are some ideas to get you started:

Things you should try:

- Filter size: Above we used 7x7; this makes pretty pictures but smaller filters may be more efficient
- Number of filters: Above we used 32 filters. Do more or fewer do better?
- Batch normalization: Try adding spatial batch normalization after convolution layers and vanilla batch normalization aafter affine layers. Do your networks train faster?
- Network architecture: The network above has two layers of trainable parameters. Can you do better with a deeper network? You can implement alternative architectures in the file `cs6353/classifiers/convnet.py`. Some good architectures to try include:
  - [conv-relu-pool]xN - conv - relu - [affine]xM - [softmax or SVM]
  - [conv-relu-pool]XN - [affine]XM - [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN - [affine]xM - [softmax or SVM]

Tips for training

For each network architecture that you try, you should tune the learning rate and regularization strength. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the course-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.

Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these; however they would be good things to try for extra credit.

- Alternative update steps: For the assignment we implemented SGD+momentum, RMSprop, and Adam; you could try alternatives like AdaGrad or AdaDelta.
- Alternative activation functions such as leaky ReLU, parametric ReLU, or MaxOut.
- Model ensembles
- Data augmentation

If you do decide to implement something extra, clearly describe it in the "Extra Credit Description" cell below.

What we expect

At the very least, you should be able to train a ConvNet that gets at least 65% accuracy on the validation set. This is just a lower bound - if you are careful it should be possible to get accuracies much higher than that! Extra credit points will be awarded for particularly high-scoring models or unique approaches.

You should use the space below to experiment and train your network. The final cell in this notebook should contain the training, validation, and test set accuracies for your final trained network. In this notebook you should also write an explanation of what you did, any additional features that you implemented, and any visualizations or graphs that you make in the process of training and evaluating your network.

Have fun and happy training!

Explanation

1. Added the convnet.py class.

- Added a conv_relu_forward layer to the ConvNet. (location: cs6353/classifiers/convnet.py)

2. Experimented with several hyper-parameter settings.
3. Experimented with adaptive learning rate instead of a static learning rate.
4. Experimented with averaging the best models and using the averaged model for prediction. Although this did not give the best results.

In [26]:
```
# Your code goes here!

# from cs6353.classifiers.cnn import *
from cs6353.classifiers.convnet import *
np.random.seed(10)


# With Seed as 10:
# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=1
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.515000; val_acc: 0.523000
```

```
# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=6
#                       filter_size=7)
# (Epoch 1 / 1) train acc: 0.398000; val_acc: 0.409000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=6
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.474000; val_acc: 0.500000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=3
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.491000; val_acc: 0.501000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=8
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.515000; val_acc: 0.507000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=1
#                       filter_size=5)
# (Epoch 1 / 1) train acc: 0.538000; val_acc: 0.496000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.001, num_filters=1
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.515000; val_acc: 0.523000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.0005, num_filters=
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.500000; val_acc: 0.480000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.005, num_filters=1
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.529000; val_acc: 0.506000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.003, num_filters=1
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.532000; val_acc: 0.522000

# model = CustomConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=1
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.526000; val_acc: 0.532000

# model = ConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=16,
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.526000; val_acc: 0.532000

# New Model:
# model = ConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=16,
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.501000; val_acc: 0.519000

# batch_size = 500
# model = ConvNet(weight_scale=0.001, hidden_dim=100, reg=0.002, num_filters=64,
#                       filter_size=3)
# (Epoch 1 / 1) train acc: 0.508000; val_acc: 0.517000

# model = ConvNet(weight_scale=0.001, hidden_dim=100, reg=0.002, num_filters=16,
```

```
#                         filter_size=7)
# (Epoch 1 / 1) train acc: 0.477000; val_acc: 0.479000

# model = ConvNet(weight_scale=0.001, hidden_dim=100, reg=0.0, num_filters=16,
#                         filter_size=3)
# (Epoch 1 / 1) train acc: 0.443000; val_acc: 0.448000

# batch_size = 50
model = ConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=16,
                        filter_size=3)

solver = Solver(model, data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 9800) loss: 2.302793
(Epoch 0 / 10) train acc: 0.112000; val_acc: 0.112000
(Iteration 21 / 9800) loss: 2.173255
(Iteration 41 / 9800) loss: 2.119265
(Iteration 61 / 9800) loss: 2.100458
(Iteration 81 / 9800) loss: 2.001850
(Iteration 101 / 9800) loss: 2.037317
(Iteration 121 / 9800) loss: 1.933021
(Iteration 141 / 9800) loss: 1.774890
(Iteration 161 / 9800) loss: 1.802223
(Iteration 181 / 9800) loss: 1.837038
(Iteration 201 / 9800) loss: 1.809888
(Iteration 221 / 9800) loss: 1.692987
(Iteration 241 / 9800) loss: 1.656548
(Iteration 261 / 9800) loss: 1.855614
(Iteration 281 / 9800) loss: 1.580088
(Iteration 301 / 9800) loss: 1.389625
(Iteration 321 / 9800) loss: 1.568423
(Iteration 341 / 9800) loss: 1.603689
(Iteration 361 / 9800) loss: 1.546651
(Iteration 381 / 9800) loss: 1.717205
(Iteration 401 / 9800) loss: 1.527824
(Iteration 421 / 9800) loss: 1.546658
(Iteration 441 / 9800) loss: 1.855837
(Iteration 461 / 9800) loss: 1.688197
(Iteration 481 / 9800) loss: 1.429002
(Iteration 501 / 9800) loss: 1.344175
(Iteration 521 / 9800) loss: 1.658252
(Iteration 541 / 9800) loss: 1.370669
(Iteration 561 / 9800) loss: 1.756421
(Iteration 581 / 9800) loss: 1.365865
(Iteration 601 / 9800) loss: 1.462633
(Iteration 621 / 9800) loss: 1.467102
(Iteration 641 / 9800) loss: 1.571033
(Iteration 661 / 9800) loss: 1.511263
(Iteration 681 / 9800) loss: 1.361913
(Iteration 701 / 9800) loss: 1.425568
(Iteration 721 / 9800) loss: 1.449560
(Iteration 741 / 9800) loss: 1.441322
(Iteration 761 / 9800) loss: 1.318328
(Iteration 781 / 9800) loss: 1.510555
(Iteration 801 / 9800) loss: 1.270371
(Iteration 821 / 9800) loss: 1.123053
(Iteration 841 / 9800) loss: 1.441036
(Iteration 861 / 9800) loss: 1.415209
(Iteration 881 / 9800) loss: 1.418594
(Iteration 901 / 9800) loss: 1.498783
(Iteration 921 / 9800) loss: 1.233422
(Iteration 941 / 9800) loss: 1.417534
(Iteration 961 / 9800) loss: 1.180133
(Epoch 1 / 10) train acc: 0.501000; val_acc: 0.519000
(Iteration 981 / 9800) loss: 1.244650
(Iteration 1001 / 9800) loss: 1.467469
(Iteration 1021 / 9800) loss: 1.260026
(Iteration 1041 / 9800) loss: 1.361507
(Iteration 1061 / 9800) loss: 1.524251
```

```
(Iteration 1081 / 9800) loss: 1.526833
(Iteration 1101 / 9800) loss: 1.723687
(Iteration 1121 / 9800) loss: 1.451566
(Iteration 1141 / 9800) loss: 1.425006
(Iteration 1161 / 9800) loss: 1.114470
(Iteration 1181 / 9800) loss: 1.259041
(Iteration 1201 / 9800) loss: 1.328284
(Iteration 1221 / 9800) loss: 1.318428
(Iteration 1241 / 9800) loss: 1.489715
(Iteration 1261 / 9800) loss: 1.353383
(Iteration 1281 / 9800) loss: 1.454362
(Iteration 1301 / 9800) loss: 1.330235
(Iteration 1321 / 9800) loss: 1.419498
(Iteration 1341 / 9800) loss: 1.015193
(Iteration 1361 / 9800) loss: 1.187602
(Iteration 1381 / 9800) loss: 1.451762
(Iteration 1401 / 9800) loss: 1.120701
(Iteration 1421 / 9800) loss: 1.345581
(Iteration 1441 / 9800) loss: 1.285713
(Iteration 1461 / 9800) loss: 1.145375
(Iteration 1481 / 9800) loss: 1.393392
(Iteration 1501 / 9800) loss: 1.417904
(Iteration 1521 / 9800) loss: 1.306106
(Iteration 1541 / 9800) loss: 1.299586
(Iteration 1561 / 9800) loss: 1.199667
(Iteration 1581 / 9800) loss: 1.418143
(Iteration 1601 / 9800) loss: 1.192813
(Iteration 1621 / 9800) loss: 1.603148
(Iteration 1641 / 9800) loss: 1.310456
(Iteration 1661 / 9800) loss: 1.318794
(Iteration 1681 / 9800) loss: 1.293175
(Iteration 1701 / 9800) loss: 1.129695
(Iteration 1721 / 9800) loss: 1.031695
(Iteration 1741 / 9800) loss: 1.195144
(Iteration 1761 / 9800) loss: 1.146562
(Iteration 1781 / 9800) loss: 1.249772
(Iteration 1801 / 9800) loss: 1.061854
(Iteration 1821 / 9800) loss: 1.274369
(Iteration 1841 / 9800) loss: 1.132488
(Iteration 1861 / 9800) loss: 1.284558
(Iteration 1881 / 9800) loss: 1.290826
(Iteration 1901 / 9800) loss: 1.293999
(Iteration 1921 / 9800) loss: 1.445781
(Iteration 1941 / 9800) loss: 1.269942
(Epoch 2 / 10) train acc: 0.554000; val_acc: 0.540000
(Iteration 1961 / 9800) loss: 0.956029
(Iteration 1981 / 9800) loss: 1.212806
(Iteration 2001 / 9800) loss: 1.108970
(Iteration 2021 / 9800) loss: 1.025524
(Iteration 2041 / 9800) loss: 1.316381
(Iteration 2061 / 9800) loss: 1.241121
(Iteration 2081 / 9800) loss: 1.070255
(Iteration 2101 / 9800) loss: 1.165151
(Iteration 2121 / 9800) loss: 1.097693
(Iteration 2141 / 9800) loss: 1.322712
(Iteration 2161 / 9800) loss: 1.299791
```

```
(Iteration 2181 / 9800) loss: 1.232600
(Iteration 2201 / 9800) loss: 1.239755
(Iteration 2221 / 9800) loss: 1.355508
(Iteration 2241 / 9800) loss: 1.037254
(Iteration 2261 / 9800) loss: 1.119363
(Iteration 2281 / 9800) loss: 1.160338
(Iteration 2301 / 9800) loss: 1.337258
(Iteration 2321 / 9800) loss: 1.303571
(Iteration 2341 / 9800) loss: 1.069354
(Iteration 2361 / 9800) loss: 1.266042
(Iteration 2381 / 9800) loss: 1.285066
(Iteration 2401 / 9800) loss: 0.993921
(Iteration 2421 / 9800) loss: 1.024891
(Iteration 2441 / 9800) loss: 1.132730
(Iteration 2461 / 9800) loss: 1.016457
(Iteration 2481 / 9800) loss: 1.021221
(Iteration 2501 / 9800) loss: 0.915766
(Iteration 2521 / 9800) loss: 1.356944
(Iteration 2541 / 9800) loss: 0.824588
(Iteration 2561 / 9800) loss: 1.051867
(Iteration 2581 / 9800) loss: 1.101769
(Iteration 2601 / 9800) loss: 1.286594
(Iteration 2621 / 9800) loss: 1.336482
(Iteration 2641 / 9800) loss: 1.141331
(Iteration 2661 / 9800) loss: 0.985529
(Iteration 2681 / 9800) loss: 1.165071
(Iteration 2701 / 9800) loss: 1.149567
(Iteration 2721 / 9800) loss: 1.260962
(Iteration 2741 / 9800) loss: 0.920081
(Iteration 2761 / 9800) loss: 1.149269
(Iteration 2781 / 9800) loss: 1.234474
(Iteration 2801 / 9800) loss: 1.136933
(Iteration 2821 / 9800) loss: 1.176206
(Iteration 2841 / 9800) loss: 1.178619
(Iteration 2861 / 9800) loss: 1.395798
(Iteration 2881 / 9800) loss: 0.980180
(Iteration 2901 / 9800) loss: 1.416911
(Iteration 2921 / 9800) loss: 1.502400
(Epoch 3 / 10) train acc: 0.628000; val_acc: 0.583000
(Iteration 2941 / 9800) loss: 1.100902
(Iteration 2961 / 9800) loss: 1.068684
(Iteration 2981 / 9800) loss: 0.799491
(Iteration 3001 / 9800) loss: 1.256800
(Iteration 3021 / 9800) loss: 0.887862
(Iteration 3041 / 9800) loss: 1.126398
(Iteration 3061 / 9800) loss: 1.251782
(Iteration 3081 / 9800) loss: 1.242703
(Iteration 3101 / 9800) loss: 1.087355
(Iteration 3121 / 9800) loss: 0.831981
(Iteration 3141 / 9800) loss: 1.432618
(Iteration 3161 / 9800) loss: 1.026325
(Iteration 3181 / 9800) loss: 1.266448
(Iteration 3201 / 9800) loss: 1.259013
(Iteration 3221 / 9800) loss: 1.383393
(Iteration 3241 / 9800) loss: 1.147623
(Iteration 3261 / 9800) loss: 1.048333
```

```
(Iteration 3281 / 9800) loss: 0.932825
(Iteration 3301 / 9800) loss: 1.431048
(Iteration 3321 / 9800) loss: 1.180083
(Iteration 3341 / 9800) loss: 1.005311
(Iteration 3361 / 9800) loss: 1.109716
(Iteration 3381 / 9800) loss: 1.206279
(Iteration 3401 / 9800) loss: 1.169828
(Iteration 3421 / 9800) loss: 1.094850
(Iteration 3441 / 9800) loss: 1.051395
(Iteration 3461 / 9800) loss: 1.158265
(Iteration 3481 / 9800) loss: 0.962507
(Iteration 3501 / 9800) loss: 0.891350
(Iteration 3521 / 9800) loss: 1.262171
(Iteration 3541 / 9800) loss: 0.966207
(Iteration 3561 / 9800) loss: 0.930752
(Iteration 3581 / 9800) loss: 0.861989
(Iteration 3601 / 9800) loss: 1.058228
(Iteration 3621 / 9800) loss: 1.111204
(Iteration 3641 / 9800) loss: 1.130015
(Iteration 3661 / 9800) loss: 1.179004
(Iteration 3681 / 9800) loss: 0.921456
(Iteration 3701 / 9800) loss: 1.401984
(Iteration 3721 / 9800) loss: 1.077891
(Iteration 3741 / 9800) loss: 0.947892
(Iteration 3761 / 9800) loss: 1.092024
(Iteration 3781 / 9800) loss: 1.276462
(Iteration 3801 / 9800) loss: 1.142631
(Iteration 3821 / 9800) loss: 1.178299
(Iteration 3841 / 9800) loss: 1.169240
(Iteration 3861 / 9800) loss: 1.152552
(Iteration 3881 / 9800) loss: 1.035562
(Iteration 3901 / 9800) loss: 1.179793
(Epoch 4 / 10) train acc: 0.677000; val_acc: 0.578000
(Iteration 3921 / 9800) loss: 1.193603
(Iteration 3941 / 9800) loss: 0.949534
(Iteration 3961 / 9800) loss: 0.973170
(Iteration 3981 / 9800) loss: 1.219043
(Iteration 4001 / 9800) loss: 1.345212
(Iteration 4021 / 9800) loss: 0.989542
(Iteration 4041 / 9800) loss: 1.170654
(Iteration 4061 / 9800) loss: 1.190251
(Iteration 4081 / 9800) loss: 0.925731
(Iteration 4101 / 9800) loss: 1.116045
(Iteration 4121 / 9800) loss: 1.188698
(Iteration 4141 / 9800) loss: 0.988667
(Iteration 4161 / 9800) loss: 0.846799
(Iteration 4181 / 9800) loss: 0.988305
(Iteration 4201 / 9800) loss: 1.236648
(Iteration 4221 / 9800) loss: 0.959893
(Iteration 4241 / 9800) loss: 1.166633
(Iteration 4261 / 9800) loss: 0.979977
(Iteration 4281 / 9800) loss: 1.196552
(Iteration 4301 / 9800) loss: 1.159437
(Iteration 4321 / 9800) loss: 1.058151
(Iteration 4341 / 9800) loss: 1.020620
(Iteration 4361 / 9800) loss: 1.223172
```

```
(Iteration 4381 / 9800) loss: 1.033470
(Iteration 4401 / 9800) loss: 0.871062
(Iteration 4421 / 9800) loss: 0.939815
(Iteration 4441 / 9800) loss: 0.873343
(Iteration 4461 / 9800) loss: 1.051496
(Iteration 4481 / 9800) loss: 1.057787
(Iteration 4501 / 9800) loss: 1.279294
(Iteration 4521 / 9800) loss: 0.790180
(Iteration 4541 / 9800) loss: 1.085596
(Iteration 4561 / 9800) loss: 1.049011
(Iteration 4581 / 9800) loss: 1.140149
(Iteration 4601 / 9800) loss: 0.946775
(Iteration 4621 / 9800) loss: 1.119058
(Iteration 4641 / 9800) loss: 1.230681
(Iteration 4661 / 9800) loss: 1.054967
(Iteration 4681 / 9800) loss: 1.103472
(Iteration 4701 / 9800) loss: 1.127262
(Iteration 4721 / 9800) loss: 1.048037
(Iteration 4741 / 9800) loss: 0.838217
(Iteration 4761 / 9800) loss: 1.307643
(Iteration 4781 / 9800) loss: 1.442733
(Iteration 4801 / 9800) loss: 1.379413
(Iteration 4821 / 9800) loss: 1.087158
(Iteration 4841 / 9800) loss: 1.022234
(Iteration 4861 / 9800) loss: 1.056968
(Iteration 4881 / 9800) loss: 1.038787
(Epoch 5 / 10) train acc: 0.715000; val_acc: 0.597000
(Iteration 4901 / 9800) loss: 1.342138
(Iteration 4921 / 9800) loss: 0.969025
(Iteration 4941 / 9800) loss: 0.875680
(Iteration 4961 / 9800) loss: 1.351256
(Iteration 4981 / 9800) loss: 0.811148
(Iteration 5001 / 9800) loss: 0.910345
(Iteration 5021 / 9800) loss: 1.028454
(Iteration 5041 / 9800) loss: 0.749576
(Iteration 5061 / 9800) loss: 1.038117
(Iteration 5081 / 9800) loss: 1.295043
(Iteration 5101 / 9800) loss: 0.771571
(Iteration 5121 / 9800) loss: 0.995872
(Iteration 5141 / 9800) loss: 0.980136
(Iteration 5161 / 9800) loss: 1.212226
(Iteration 5181 / 9800) loss: 0.754605
(Iteration 5201 / 9800) loss: 1.100554
(Iteration 5221 / 9800) loss: 1.159101
(Iteration 5241 / 9800) loss: 1.281673
(Iteration 5261 / 9800) loss: 1.301016
(Iteration 5281 / 9800) loss: 1.026797
(Iteration 5301 / 9800) loss: 1.066406
(Iteration 5321 / 9800) loss: 0.741331
(Iteration 5341 / 9800) loss: 0.743554
(Iteration 5361 / 9800) loss: 1.201698
(Iteration 5381 / 9800) loss: 1.092635
(Iteration 5401 / 9800) loss: 1.058120
(Iteration 5421 / 9800) loss: 0.811247
(Iteration 5441 / 9800) loss: 0.850032
(Iteration 5461 / 9800) loss: 0.929833
```

```
(Iteration 5481 / 9800) loss: 1.022258
(Iteration 5501 / 9800) loss: 0.959697
(Iteration 5521 / 9800) loss: 0.844392
(Iteration 5541 / 9800) loss: 0.849220
(Iteration 5561 / 9800) loss: 1.097020
(Iteration 5581 / 9800) loss: 1.355168
(Iteration 5601 / 9800) loss: 0.800781
(Iteration 5621 / 9800) loss: 0.962723
(Iteration 5641 / 9800) loss: 1.134219
(Iteration 5661 / 9800) loss: 0.919175
(Iteration 5681 / 9800) loss: 0.918215
(Iteration 5701 / 9800) loss: 1.197993
(Iteration 5721 / 9800) loss: 0.866036
(Iteration 5741 / 9800) loss: 0.999217
(Iteration 5761 / 9800) loss: 1.247881
(Iteration 5781 / 9800) loss: 0.903123
(Iteration 5801 / 9800) loss: 1.040367
(Iteration 5821 / 9800) loss: 0.914202
(Iteration 5841 / 9800) loss: 1.002449
(Iteration 5861 / 9800) loss: 1.004899
(Epoch 6 / 10) train acc: 0.716000; val_acc: 0.619000
(Iteration 5881 / 9800) loss: 1.127592
(Iteration 5901 / 9800) loss: 1.053831
(Iteration 5921 / 9800) loss: 0.904439
(Iteration 5941 / 9800) loss: 0.930834
(Iteration 5961 / 9800) loss: 0.994450
(Iteration 5981 / 9800) loss: 0.865699
(Iteration 6001 / 9800) loss: 0.846896
(Iteration 6021 / 9800) loss: 1.086112
(Iteration 6041 / 9800) loss: 0.957196
(Iteration 6061 / 9800) loss: 0.979015
(Iteration 6081 / 9800) loss: 0.853114
(Iteration 6101 / 9800) loss: 1.118740
(Iteration 6121 / 9800) loss: 1.001317
(Iteration 6141 / 9800) loss: 1.026994
(Iteration 6161 / 9800) loss: 1.087257
(Iteration 6181 / 9800) loss: 0.758304
(Iteration 6201 / 9800) loss: 0.920684
(Iteration 6221 / 9800) loss: 1.252425
(Iteration 6241 / 9800) loss: 1.010692
(Iteration 6261 / 9800) loss: 0.782677
(Iteration 6281 / 9800) loss: 0.915362
(Iteration 6301 / 9800) loss: 0.828845
(Iteration 6321 / 9800) loss: 0.925772
(Iteration 6341 / 9800) loss: 1.193388
(Iteration 6361 / 9800) loss: 1.005088
(Iteration 6381 / 9800) loss: 0.968746
(Iteration 6401 / 9800) loss: 1.014393
(Iteration 6421 / 9800) loss: 1.122584
(Iteration 6441 / 9800) loss: 0.854857
(Iteration 6461 / 9800) loss: 0.794962
(Iteration 6481 / 9800) loss: 0.957128
(Iteration 6501 / 9800) loss: 1.083102
(Iteration 6521 / 9800) loss: 1.169035
(Iteration 6541 / 9800) loss: 0.969143
(Iteration 6561 / 9800) loss: 1.040095
```

```
(Iteration 6581 / 9800) loss: 1.067673
(Iteration 6601 / 9800) loss: 1.049297
(Iteration 6621 / 9800) loss: 1.109505
(Iteration 6641 / 9800) loss: 0.939858
(Iteration 6661 / 9800) loss: 1.048938
(Iteration 6681 / 9800) loss: 1.114083
(Iteration 6701 / 9800) loss: 0.947882
(Iteration 6721 / 9800) loss: 1.082782
(Iteration 6741 / 9800) loss: 0.760445
(Iteration 6761 / 9800) loss: 0.800399
(Iteration 6781 / 9800) loss: 1.181059
(Iteration 6801 / 9800) loss: 1.234802
(Iteration 6821 / 9800) loss: 0.837882
(Iteration 6841 / 9800) loss: 0.909873
(Epoch 7 / 10) train acc: 0.690000; val_acc: 0.636000
(Iteration 6861 / 9800) loss: 0.890018
(Iteration 6881 / 9800) loss: 1.069382
(Iteration 6901 / 9800) loss: 0.770274
(Iteration 6921 / 9800) loss: 1.010613
(Iteration 6941 / 9800) loss: 0.826196
(Iteration 6961 / 9800) loss: 1.183044
(Iteration 6981 / 9800) loss: 0.931142
(Iteration 7001 / 9800) loss: 0.962308
(Iteration 7021 / 9800) loss: 0.687085
(Iteration 7041 / 9800) loss: 0.766227
(Iteration 7061 / 9800) loss: 0.997663
(Iteration 7081 / 9800) loss: 0.773421
(Iteration 7101 / 9800) loss: 0.991770
(Iteration 7121 / 9800) loss: 0.950112
(Iteration 7141 / 9800) loss: 0.992526
(Iteration 7161 / 9800) loss: 1.402810
(Iteration 7181 / 9800) loss: 0.924557
(Iteration 7201 / 9800) loss: 1.111236
(Iteration 7221 / 9800) loss: 1.025470
(Iteration 7241 / 9800) loss: 1.040674
(Iteration 7261 / 9800) loss: 1.053129
(Iteration 7281 / 9800) loss: 1.113131
(Iteration 7301 / 9800) loss: 0.874932
(Iteration 7321 / 9800) loss: 0.824890
(Iteration 7341 / 9800) loss: 1.073680
(Iteration 7361 / 9800) loss: 1.000575
(Iteration 7381 / 9800) loss: 0.785925
(Iteration 7401 / 9800) loss: 0.952165
(Iteration 7421 / 9800) loss: 0.911277
(Iteration 7441 / 9800) loss: 0.936853
(Iteration 7461 / 9800) loss: 1.171728
(Iteration 7481 / 9800) loss: 0.898757
(Iteration 7501 / 9800) loss: 1.177108
(Iteration 7521 / 9800) loss: 0.879987
(Iteration 7541 / 9800) loss: 1.018334
(Iteration 7561 / 9800) loss: 1.198725
(Iteration 7581 / 9800) loss: 0.924463
(Iteration 7601 / 9800) loss: 0.954736
(Iteration 7621 / 9800) loss: 0.965663
(Iteration 7641 / 9800) loss: 0.920204
(Iteration 7661 / 9800) loss: 0.688926
```

```
(Iteration 7681 / 9800) loss: 0.953068
(Iteration 7701 / 9800) loss: 0.842252
(Iteration 7721 / 9800) loss: 1.223962
(Iteration 7741 / 9800) loss: 1.318182
(Iteration 7761 / 9800) loss: 1.020376
(Iteration 7781 / 9800) loss: 0.793083
(Iteration 7801 / 9800) loss: 0.779986
(Iteration 7821 / 9800) loss: 0.808837
(Epoch 8 / 10) train acc: 0.743000; val_acc: 0.638000
(Iteration 7841 / 9800) loss: 0.958436
(Iteration 7861 / 9800) loss: 0.767437
(Iteration 7881 / 9800) loss: 1.157341
(Iteration 7901 / 9800) loss: 0.929333
(Iteration 7921 / 9800) loss: 0.777333
(Iteration 7941 / 9800) loss: 0.832997
(Iteration 7961 / 9800) loss: 1.121416
(Iteration 7981 / 9800) loss: 0.893870
(Iteration 8001 / 9800) loss: 1.096410
(Iteration 8021 / 9800) loss: 0.853013
(Iteration 8041 / 9800) loss: 0.875890
(Iteration 8061 / 9800) loss: 1.082447
(Iteration 8081 / 9800) loss: 0.944758
(Iteration 8101 / 9800) loss: 0.770966
(Iteration 8121 / 9800) loss: 0.749308
(Iteration 8141 / 9800) loss: 1.274906
(Iteration 8161 / 9800) loss: 0.970191
(Iteration 8181 / 9800) loss: 0.537531
(Iteration 8201 / 9800) loss: 0.797611
(Iteration 8221 / 9800) loss: 0.734632
(Iteration 8241 / 9800) loss: 0.839868
(Iteration 8261 / 9800) loss: 1.103837
(Iteration 8281 / 9800) loss: 0.913491
(Iteration 8301 / 9800) loss: 1.246221
(Iteration 8321 / 9800) loss: 0.650573
(Iteration 8341 / 9800) loss: 0.779492
(Iteration 8361 / 9800) loss: 0.997751
(Iteration 8381 / 9800) loss: 0.853772
(Iteration 8401 / 9800) loss: 1.023431
(Iteration 8421 / 9800) loss: 1.050506
(Iteration 8441 / 9800) loss: 1.090220
(Iteration 8461 / 9800) loss: 0.849587
(Iteration 8481 / 9800) loss: 0.704686
(Iteration 8501 / 9800) loss: 0.959704
(Iteration 8521 / 9800) loss: 0.822649
(Iteration 8541 / 9800) loss: 1.183581
(Iteration 8561 / 9800) loss: 0.797522
(Iteration 8581 / 9800) loss: 0.897489
(Iteration 8601 / 9800) loss: 0.836869
(Iteration 8621 / 9800) loss: 0.857317
(Iteration 8641 / 9800) loss: 0.922509
(Iteration 8661 / 9800) loss: 0.893841
(Iteration 8681 / 9800) loss: 1.031013
(Iteration 8701 / 9800) loss: 0.929120
(Iteration 8721 / 9800) loss: 1.076772
(Iteration 8741 / 9800) loss: 1.004324
(Iteration 8761 / 9800) loss: 1.012801
```

```
(Iteration 8781 / 9800) loss: 0.739718
(Iteration 8801 / 9800) loss: 0.772395
(Epoch 9 / 10) train acc: 0.718000; val_acc: 0.636000
(Iteration 8821 / 9800) loss: 0.920763
(Iteration 8841 / 9800) loss: 0.548163
(Iteration 8861 / 9800) loss: 0.714371
(Iteration 8881 / 9800) loss: 0.911710
(Iteration 8901 / 9800) loss: 1.014152
(Iteration 8921 / 9800) loss: 0.942840
(Iteration 8941 / 9800) loss: 0.587880
(Iteration 8961 / 9800) loss: 0.894132
(Iteration 8981 / 9800) loss: 0.863635
(Iteration 9001 / 9800) loss: 0.706373
(Iteration 9021 / 9800) loss: 1.023914
(Iteration 9041 / 9800) loss: 1.061046
(Iteration 9061 / 9800) loss: 1.036501
(Iteration 9081 / 9800) loss: 1.099908
(Iteration 9101 / 9800) loss: 0.739478
(Iteration 9121 / 9800) loss: 0.838970
(Iteration 9141 / 9800) loss: 1.115781
(Iteration 9161 / 9800) loss: 1.028451
(Iteration 9181 / 9800) loss: 0.954352
(Iteration 9201 / 9800) loss: 0.977249
(Iteration 9221 / 9800) loss: 1.176499
(Iteration 9241 / 9800) loss: 0.905785
(Iteration 9261 / 9800) loss: 1.003460
(Iteration 9281 / 9800) loss: 0.917962
(Iteration 9301 / 9800) loss: 0.893511
(Iteration 9321 / 9800) loss: 0.851737
(Iteration 9341 / 9800) loss: 0.790723
(Iteration 9361 / 9800) loss: 0.822350
(Iteration 9381 / 9800) loss: 0.695687
(Iteration 9401 / 9800) loss: 0.659990
(Iteration 9421 / 9800) loss: 1.020467
(Iteration 9441 / 9800) loss: 0.850126
(Iteration 9461 / 9800) loss: 0.954112
(Iteration 9481 / 9800) loss: 0.849583
(Iteration 9501 / 9800) loss: 1.104088
(Iteration 9521 / 9800) loss: 0.943589
(Iteration 9541 / 9800) loss: 1.171611
(Iteration 9561 / 9800) loss: 1.102860
(Iteration 9581 / 9800) loss: 0.790230
(Iteration 9601 / 9800) loss: 0.524208
(Iteration 9621 / 9800) loss: 1.043301
(Iteration 9641 / 9800) loss: 1.008132
(Iteration 9661 / 9800) loss: 0.799770
(Iteration 9681 / 9800) loss: 0.619169
(Iteration 9701 / 9800) loss: 0.923397
(Iteration 9721 / 9800) loss: 0.960808
(Iteration 9741 / 9800) loss: 1.040369
(Iteration 9761 / 9800) loss: 1.227906
(Iteration 9781 / 9800) loss: 0.797653
(Epoch 10 / 10) train acc: 0.758000; val_acc: 0.625000
```

In [28]:
```
import copy
```

```
trained_model_1 = copy.deepcopy(model)

solver.num_epochs = 5
solver.optim_config['learning_rate'] = 1e-4
solver.train()
```

```
(Iteration 1 / 4900) loss: 0.962498
(Epoch 10 / 5) train acc: 0.756000; val_acc: 0.639000
(Iteration 21 / 4900) loss: 0.940863
(Iteration 41 / 4900) loss: 1.177390
(Iteration 61 / 4900) loss: 1.240448
(Iteration 81 / 4900) loss: 0.937390
(Iteration 101 / 4900) loss: 1.165239
(Iteration 121 / 4900) loss: 0.943411
(Iteration 141 / 4900) loss: 0.776808
(Iteration 161 / 4900) loss: 0.893728
(Iteration 181 / 4900) loss: 0.913712
(Iteration 201 / 4900) loss: 0.909020
(Iteration 221 / 4900) loss: 0.834575
(Iteration 241 / 4900) loss: 1.049193
(Iteration 261 / 4900) loss: 1.113575
(Iteration 281 / 4900) loss: 0.786026
(Iteration 301 / 4900) loss: 0.751560
(Iteration 321 / 4900) loss: 1.043334
(Iteration 341 / 4900) loss: 0.845228
(Iteration 361 / 4900) loss: 1.070011
(Iteration 381 / 4900) loss: 0.735466
(Iteration 401 / 4900) loss: 1.100376
(Iteration 421 / 4900) loss: 0.771272
(Iteration 441 / 4900) loss: 0.978428
(Iteration 461 / 4900) loss: 0.816013
(Iteration 481 / 4900) loss: 0.785697
(Iteration 501 / 4900) loss: 0.875705
(Iteration 521 / 4900) loss: 0.595900
(Iteration 541 / 4900) loss: 1.110664
(Iteration 561 / 4900) loss: 0.878782
(Iteration 581 / 4900) loss: 0.750858
(Iteration 601 / 4900) loss: 0.973598
(Iteration 621 / 4900) loss: 1.020268
(Iteration 641 / 4900) loss: 0.954053
(Iteration 661 / 4900) loss: 0.939527
(Iteration 681 / 4900) loss: 1.034724
(Iteration 701 / 4900) loss: 1.065731
(Iteration 721 / 4900) loss: 1.108514
(Iteration 741 / 4900) loss: 0.927374
(Iteration 761 / 4900) loss: 0.953373
(Iteration 781 / 4900) loss: 0.729724
(Iteration 801 / 4900) loss: 0.719359
(Iteration 821 / 4900) loss: 1.108262
(Iteration 841 / 4900) loss: 1.087829
(Iteration 861 / 4900) loss: 1.057824
(Iteration 881 / 4900) loss: 0.920874
(Iteration 901 / 4900) loss: 0.833016
(Iteration 921 / 4900) loss: 0.903550
(Iteration 941 / 4900) loss: 1.089947
(Iteration 961 / 4900) loss: 0.831852
(Epoch 11 / 5) train acc: 0.754000; val_acc: 0.634000
(Iteration 981 / 4900) loss: 0.887652
(Iteration 1001 / 4900) loss: 0.933843
(Iteration 1021 / 4900) loss: 0.565867
(Iteration 1041 / 4900) loss: 0.996721
(Iteration 1061 / 4900) loss: 0.928410
```

```
(Iteration 1081 / 4900) loss: 0.906726
(Iteration 1101 / 4900) loss: 1.154408
(Iteration 1121 / 4900) loss: 0.832301
(Iteration 1141 / 4900) loss: 0.828680
(Iteration 1161 / 4900) loss: 0.926402
(Iteration 1181 / 4900) loss: 0.844428
(Iteration 1201 / 4900) loss: 0.784300
(Iteration 1221 / 4900) loss: 0.904652
(Iteration 1241 / 4900) loss: 0.823114
(Iteration 1261 / 4900) loss: 0.987984
(Iteration 1281 / 4900) loss: 1.161680
(Iteration 1301 / 4900) loss: 0.756356
(Iteration 1321 / 4900) loss: 0.659235
(Iteration 1341 / 4900) loss: 1.017696
(Iteration 1361 / 4900) loss: 0.904077
(Iteration 1381 / 4900) loss: 0.832376
(Iteration 1401 / 4900) loss: 0.988413
(Iteration 1421 / 4900) loss: 0.706359
(Iteration 1441 / 4900) loss: 1.201010
(Iteration 1461 / 4900) loss: 1.054737
(Iteration 1481 / 4900) loss: 1.192820
(Iteration 1501 / 4900) loss: 0.998209
(Iteration 1521 / 4900) loss: 1.054879
(Iteration 1541 / 4900) loss: 0.909531
(Iteration 1561 / 4900) loss: 0.760340
(Iteration 1581 / 4900) loss: 0.953680
(Iteration 1601 / 4900) loss: 0.952818
(Iteration 1621 / 4900) loss: 0.643169
(Iteration 1641 / 4900) loss: 0.851851
(Iteration 1661 / 4900) loss: 0.998477
(Iteration 1681 / 4900) loss: 0.879551
(Iteration 1701 / 4900) loss: 0.959569
(Iteration 1721 / 4900) loss: 1.214331
(Iteration 1741 / 4900) loss: 0.710017
(Iteration 1761 / 4900) loss: 0.686616
(Iteration 1781 / 4900) loss: 0.763493
(Iteration 1801 / 4900) loss: 0.625088
(Iteration 1821 / 4900) loss: 0.683973
(Iteration 1841 / 4900) loss: 0.697474
(Iteration 1861 / 4900) loss: 1.068422
(Iteration 1881 / 4900) loss: 0.872040
(Iteration 1901 / 4900) loss: 0.848058
(Iteration 1921 / 4900) loss: 0.848031
(Iteration 1941 / 4900) loss: 0.812605
(Epoch 12 / 5) train acc: 0.721000; val_acc: 0.644000
(Iteration 1961 / 4900) loss: 1.197856
(Iteration 1981 / 4900) loss: 1.029003
(Iteration 2001 / 4900) loss: 0.705205
(Iteration 2021 / 4900) loss: 0.991709
(Iteration 2041 / 4900) loss: 0.776916
(Iteration 2061 / 4900) loss: 0.777515
(Iteration 2081 / 4900) loss: 0.823748
(Iteration 2101 / 4900) loss: 0.630210
(Iteration 2121 / 4900) loss: 0.715269
(Iteration 2141 / 4900) loss: 0.869717
(Iteration 2161 / 4900) loss: 0.976635
```

```
(Iteration 2181 / 4900) loss: 0.778237
(Iteration 2201 / 4900) loss: 0.848439
(Iteration 2221 / 4900) loss: 0.978066
(Iteration 2241 / 4900) loss: 0.889973
(Iteration 2261 / 4900) loss: 1.134310
(Iteration 2281 / 4900) loss: 1.125684
(Iteration 2301 / 4900) loss: 1.125992
(Iteration 2321 / 4900) loss: 0.937474
(Iteration 2341 / 4900) loss: 0.876493
(Iteration 2361 / 4900) loss: 0.797967
(Iteration 2381 / 4900) loss: 0.866400
(Iteration 2401 / 4900) loss: 0.867201
(Iteration 2421 / 4900) loss: 0.807949
(Iteration 2441 / 4900) loss: 1.093053
(Iteration 2461 / 4900) loss: 0.980032
(Iteration 2481 / 4900) loss: 0.741070
(Iteration 2501 / 4900) loss: 0.932286
(Iteration 2521 / 4900) loss: 0.933232
(Iteration 2541 / 4900) loss: 1.044203
(Iteration 2561 / 4900) loss: 0.875130
(Iteration 2581 / 4900) loss: 0.854955
(Iteration 2601 / 4900) loss: 1.126348
(Iteration 2621 / 4900) loss: 0.760610
(Iteration 2641 / 4900) loss: 0.857569
(Iteration 2661 / 4900) loss: 0.810115
(Iteration 2681 / 4900) loss: 0.763976
(Iteration 2701 / 4900) loss: 0.658647
(Iteration 2721 / 4900) loss: 1.055919
(Iteration 2741 / 4900) loss: 1.131859
(Iteration 2761 / 4900) loss: 0.826271
(Iteration 2781 / 4900) loss: 0.869769
(Iteration 2801 / 4900) loss: 0.940138
(Iteration 2821 / 4900) loss: 0.864243
(Iteration 2841 / 4900) loss: 0.763399
(Iteration 2861 / 4900) loss: 1.035177
(Iteration 2881 / 4900) loss: 0.838353
(Iteration 2901 / 4900) loss: 1.153393
(Iteration 2921 / 4900) loss: 0.929982
(Epoch 13 / 5) train acc: 0.713000; val_acc: 0.623000
(Iteration 2941 / 4900) loss: 1.183364
(Iteration 2961 / 4900) loss: 0.897115
(Iteration 2981 / 4900) loss: 1.084527
(Iteration 3001 / 4900) loss: 1.068303
(Iteration 3021 / 4900) loss: 0.945470
(Iteration 3041 / 4900) loss: 0.985210
(Iteration 3061 / 4900) loss: 0.762901
(Iteration 3081 / 4900) loss: 0.718934
(Iteration 3101 / 4900) loss: 0.881138
(Iteration 3121 / 4900) loss: 0.766979
(Iteration 3141 / 4900) loss: 0.770300
(Iteration 3161 / 4900) loss: 0.756564
(Iteration 3181 / 4900) loss: 0.979455
(Iteration 3201 / 4900) loss: 0.707216
(Iteration 3221 / 4900) loss: 0.811314
(Iteration 3241 / 4900) loss: 0.935184
(Iteration 3261 / 4900) loss: 1.063229
```

```
(Iteration 3281 / 4900) loss: 1.137630
(Iteration 3301 / 4900) loss: 0.769913
(Iteration 3321 / 4900) loss: 0.883132
(Iteration 3341 / 4900) loss: 0.979851
(Iteration 3361 / 4900) loss: 0.681778
(Iteration 3381 / 4900) loss: 0.993619
(Iteration 3401 / 4900) loss: 1.047423
(Iteration 3421 / 4900) loss: 0.835933
(Iteration 3441 / 4900) loss: 0.808834
(Iteration 3461 / 4900) loss: 0.723625
(Iteration 3481 / 4900) loss: 0.635115
(Iteration 3501 / 4900) loss: 0.871381
(Iteration 3521 / 4900) loss: 1.186562
(Iteration 3541 / 4900) loss: 0.674681
(Iteration 3561 / 4900) loss: 0.926352
(Iteration 3581 / 4900) loss: 0.704957
(Iteration 3601 / 4900) loss: 0.901383
(Iteration 3621 / 4900) loss: 0.971682
(Iteration 3641 / 4900) loss: 0.815859
(Iteration 3661 / 4900) loss: 0.978087
(Iteration 3681 / 4900) loss: 0.657502
(Iteration 3701 / 4900) loss: 0.960285
(Iteration 3721 / 4900) loss: 1.134952
(Iteration 3741 / 4900) loss: 0.787507
(Iteration 3761 / 4900) loss: 0.817632
(Iteration 3781 / 4900) loss: 1.021205
(Iteration 3801 / 4900) loss: 0.904321
(Iteration 3821 / 4900) loss: 0.878310
(Iteration 3841 / 4900) loss: 0.625952
(Iteration 3861 / 4900) loss: 0.820868
(Iteration 3881 / 4900) loss: 0.739372
(Iteration 3901 / 4900) loss: 0.827854
(Epoch 14 / 5) train acc: 0.779000; val_acc: 0.630000
(Iteration 3921 / 4900) loss: 1.014680
(Iteration 3941 / 4900) loss: 0.768003
(Iteration 3961 / 4900) loss: 0.708178
(Iteration 3981 / 4900) loss: 0.824769
(Iteration 4001 / 4900) loss: 0.946611
(Iteration 4021 / 4900) loss: 0.680741
(Iteration 4041 / 4900) loss: 0.856742
(Iteration 4061 / 4900) loss: 1.162482
(Iteration 4081 / 4900) loss: 1.309502
(Iteration 4101 / 4900) loss: 0.981614
(Iteration 4121 / 4900) loss: 0.795746
(Iteration 4141 / 4900) loss: 1.156969
(Iteration 4161 / 4900) loss: 0.844892
(Iteration 4181 / 4900) loss: 0.925053
(Iteration 4201 / 4900) loss: 0.929932
(Iteration 4221 / 4900) loss: 0.822147
(Iteration 4241 / 4900) loss: 0.948600
(Iteration 4261 / 4900) loss: 0.962771
(Iteration 4281 / 4900) loss: 0.798800
(Iteration 4301 / 4900) loss: 0.784074
(Iteration 4321 / 4900) loss: 0.763947
(Iteration 4341 / 4900) loss: 0.927897
(Iteration 4361 / 4900) loss: 0.603395
```

```
(Iteration 4381 / 4900) loss: 1.148339
(Iteration 4401 / 4900) loss: 0.777529
(Iteration 4421 / 4900) loss: 0.871369
(Iteration 4441 / 4900) loss: 0.709997
(Iteration 4461 / 4900) loss: 0.909269
(Iteration 4481 / 4900) loss: 0.891219
(Iteration 4501 / 4900) loss: 0.980997
(Iteration 4521 / 4900) loss: 0.716844
(Iteration 4541 / 4900) loss: 0.999627
(Iteration 4561 / 4900) loss: 0.704300
(Iteration 4581 / 4900) loss: 0.901245
(Iteration 4601 / 4900) loss: 0.661223
(Iteration 4621 / 4900) loss: 1.095969
(Iteration 4641 / 4900) loss: 0.926837
(Iteration 4661 / 4900) loss: 0.842360
(Iteration 4681 / 4900) loss: 1.144351
(Iteration 4701 / 4900) loss: 0.999944
(Iteration 4721 / 4900) loss: 0.832276
(Iteration 4741 / 4900) loss: 0.810440
(Iteration 4761 / 4900) loss: 0.926996
(Iteration 4781 / 4900) loss: 0.929186
(Iteration 4801 / 4900) loss: 0.787735
(Iteration 4821 / 4900) loss: 0.819345
(Iteration 4841 / 4900) loss: 0.919968
(Iteration 4861 / 4900) loss: 0.971817
(Iteration 4881 / 4900) loss: 0.698841
(Epoch 15 / 5) train acc: 0.793000; val_acc: 0.616000
```

In [29]:
```
solver.num_epochs = 5
solver.optim_config['learning_rate'] = 1e-3
solver.train()
```

```
(Iteration 1 / 4900) loss: 0.953305
(Epoch 15 / 5) train acc: 0.762000; val_acc: 0.643000
(Iteration 21 / 4900) loss: 1.039702
(Iteration 41 / 4900) loss: 1.173204
(Iteration 61 / 4900) loss: 0.807525
(Iteration 81 / 4900) loss: 1.053811
(Iteration 101 / 4900) loss: 0.947382
(Iteration 121 / 4900) loss: 0.909172
(Iteration 141 / 4900) loss: 0.906271
(Iteration 161 / 4900) loss: 1.102247
(Iteration 181 / 4900) loss: 0.849380
(Iteration 201 / 4900) loss: 0.874348
(Iteration 221 / 4900) loss: 1.045546
(Iteration 241 / 4900) loss: 0.992896
(Iteration 261 / 4900) loss: 1.036299
(Iteration 281 / 4900) loss: 0.872371
(Iteration 301 / 4900) loss: 0.838675
(Iteration 321 / 4900) loss: 0.895729
(Iteration 341 / 4900) loss: 0.906570
(Iteration 361 / 4900) loss: 0.647878
(Iteration 381 / 4900) loss: 0.816225
(Iteration 401 / 4900) loss: 0.792763
(Iteration 421 / 4900) loss: 0.892800
(Iteration 441 / 4900) loss: 0.700033
(Iteration 461 / 4900) loss: 0.941180
(Iteration 481 / 4900) loss: 0.990882
(Iteration 501 / 4900) loss: 1.279965
(Iteration 521 / 4900) loss: 0.882075
(Iteration 541 / 4900) loss: 0.985488
(Iteration 561 / 4900) loss: 1.062086
(Iteration 581 / 4900) loss: 0.890396
(Iteration 601 / 4900) loss: 0.818870
(Iteration 621 / 4900) loss: 0.814891
(Iteration 641 / 4900) loss: 0.888304
(Iteration 661 / 4900) loss: 1.113160
(Iteration 681 / 4900) loss: 1.101730
(Iteration 701 / 4900) loss: 1.148719
(Iteration 721 / 4900) loss: 0.755562
(Iteration 741 / 4900) loss: 0.893967
(Iteration 761 / 4900) loss: 0.912679
(Iteration 781 / 4900) loss: 0.857145
(Iteration 801 / 4900) loss: 0.784564
(Iteration 821 / 4900) loss: 0.844694
(Iteration 841 / 4900) loss: 1.048107
(Iteration 861 / 4900) loss: 0.852812
(Iteration 881 / 4900) loss: 1.045336
(Iteration 901 / 4900) loss: 0.775437
(Iteration 921 / 4900) loss: 0.826932
(Iteration 941 / 4900) loss: 1.268044
(Iteration 961 / 4900) loss: 1.118985
(Epoch 16 / 5) train acc: 0.758000; val_acc: 0.620000
(Iteration 981 / 4900) loss: 0.956469
(Iteration 1001 / 4900) loss: 0.860676
(Iteration 1021 / 4900) loss: 0.896135
(Iteration 1041 / 4900) loss: 0.865689
(Iteration 1061 / 4900) loss: 0.905341
```

```
(Iteration 1081 / 4900) loss: 0.579335
(Iteration 1101 / 4900) loss: 0.807528
(Iteration 1121 / 4900) loss: 0.980266
(Iteration 1141 / 4900) loss: 1.058640
(Iteration 1161 / 4900) loss: 0.866585
(Iteration 1181 / 4900) loss: 1.018347
(Iteration 1201 / 4900) loss: 0.586665
(Iteration 1221 / 4900) loss: 0.736839
(Iteration 1241 / 4900) loss: 0.986009
(Iteration 1261 / 4900) loss: 0.740327
(Iteration 1281 / 4900) loss: 0.742374
(Iteration 1301 / 4900) loss: 1.103848
(Iteration 1321 / 4900) loss: 0.920042
(Iteration 1341 / 4900) loss: 0.799507
(Iteration 1361 / 4900) loss: 0.896366
(Iteration 1381 / 4900) loss: 0.788168
(Iteration 1401 / 4900) loss: 0.869007
(Iteration 1421 / 4900) loss: 0.892952
(Iteration 1441 / 4900) loss: 0.686822
(Iteration 1461 / 4900) loss: 0.863316
(Iteration 1481 / 4900) loss: 1.036040
(Iteration 1501 / 4900) loss: 0.984742
(Iteration 1521 / 4900) loss: 0.741364
(Iteration 1541 / 4900) loss: 0.889634
(Iteration 1561 / 4900) loss: 0.689313
(Iteration 1581 / 4900) loss: 0.887284
(Iteration 1601 / 4900) loss: 0.812556
(Iteration 1621 / 4900) loss: 0.749796
(Iteration 1641 / 4900) loss: 1.037226
(Iteration 1661 / 4900) loss: 0.860614
(Iteration 1681 / 4900) loss: 1.005780
(Iteration 1701 / 4900) loss: 0.943679
(Iteration 1721 / 4900) loss: 0.787107
(Iteration 1741 / 4900) loss: 0.936981
(Iteration 1761 / 4900) loss: 0.686407
(Iteration 1781 / 4900) loss: 1.172318
(Iteration 1801 / 4900) loss: 1.151511
(Iteration 1821 / 4900) loss: 0.744989
(Iteration 1841 / 4900) loss: 0.661505
(Iteration 1861 / 4900) loss: 0.692241
(Iteration 1881 / 4900) loss: 0.824912
(Iteration 1901 / 4900) loss: 1.471051
(Iteration 1921 / 4900) loss: 0.652780
(Iteration 1941 / 4900) loss: 0.694758
(Epoch 17 / 5) train acc: 0.768000; val_acc: 0.644000
(Iteration 1961 / 4900) loss: 0.956508
(Iteration 1981 / 4900) loss: 0.694828
(Iteration 2001 / 4900) loss: 0.703134
(Iteration 2021 / 4900) loss: 1.006091
(Iteration 2041 / 4900) loss: 0.671301
(Iteration 2061 / 4900) loss: 0.793819
(Iteration 2081 / 4900) loss: 0.938853
(Iteration 2101 / 4900) loss: 0.934988
(Iteration 2121 / 4900) loss: 0.921861
(Iteration 2141 / 4900) loss: 0.870192
(Iteration 2161 / 4900) loss: 1.030882
```

```
(Iteration 2181 / 4900) loss: 0.965847
(Iteration 2201 / 4900) loss: 0.774769
(Iteration 2221 / 4900) loss: 0.850468
(Iteration 2241 / 4900) loss: 1.161681
(Iteration 2261 / 4900) loss: 1.028417
(Iteration 2281 / 4900) loss: 0.977963
(Iteration 2301 / 4900) loss: 0.809730
(Iteration 2321 / 4900) loss: 0.795227
(Iteration 2341 / 4900) loss: 0.718215
(Iteration 2361 / 4900) loss: 0.727343
(Iteration 2381 / 4900) loss: 0.836011
(Iteration 2401 / 4900) loss: 1.055247
(Iteration 2421 / 4900) loss: 1.059050
(Iteration 2441 / 4900) loss: 0.772244
(Iteration 2461 / 4900) loss: 0.896731
(Iteration 2481 / 4900) loss: 0.757477
(Iteration 2501 / 4900) loss: 0.710074
(Iteration 2521 / 4900) loss: 0.620806
(Iteration 2541 / 4900) loss: 0.827177
(Iteration 2561 / 4900) loss: 0.795699
(Iteration 2581 / 4900) loss: 1.085501
(Iteration 2601 / 4900) loss: 0.841990
(Iteration 2621 / 4900) loss: 0.646384
(Iteration 2641 / 4900) loss: 1.097912
(Iteration 2661 / 4900) loss: 0.910457
(Iteration 2681 / 4900) loss: 0.927862
(Iteration 2701 / 4900) loss: 0.817011
(Iteration 2721 / 4900) loss: 0.870514
(Iteration 2741 / 4900) loss: 0.951395
(Iteration 2761 / 4900) loss: 0.721758
(Iteration 2781 / 4900) loss: 1.247759
(Iteration 2801 / 4900) loss: 0.718730
(Iteration 2821 / 4900) loss: 0.899005
(Iteration 2841 / 4900) loss: 0.858685
(Iteration 2861 / 4900) loss: 0.880563
(Iteration 2881 / 4900) loss: 0.837868
(Iteration 2901 / 4900) loss: 0.808484
(Iteration 2921 / 4900) loss: 0.971324
(Epoch 18 / 5) train acc: 0.776000; val_acc: 0.624000
(Iteration 2941 / 4900) loss: 0.864324
(Iteration 2961 / 4900) loss: 0.936007
(Iteration 2981 / 4900) loss: 0.713713
(Iteration 3001 / 4900) loss: 0.717906
(Iteration 3021 / 4900) loss: 0.689481
(Iteration 3041 / 4900) loss: 0.913828
(Iteration 3061 / 4900) loss: 0.858366
(Iteration 3081 / 4900) loss: 0.852966
(Iteration 3101 / 4900) loss: 0.783260
(Iteration 3121 / 4900) loss: 0.661646
(Iteration 3141 / 4900) loss: 1.022019
(Iteration 3161 / 4900) loss: 1.074252
(Iteration 3181 / 4900) loss: 0.730403
(Iteration 3201 / 4900) loss: 0.911531
(Iteration 3221 / 4900) loss: 0.882954
(Iteration 3241 / 4900) loss: 0.818666
(Iteration 3261 / 4900) loss: 0.755700
```

```
(Iteration 3281 / 4900) loss: 0.907631
(Iteration 3301 / 4900) loss: 0.994396
(Iteration 3321 / 4900) loss: 0.840712
(Iteration 3341 / 4900) loss: 0.902290
(Iteration 3361 / 4900) loss: 1.358271
(Iteration 3381 / 4900) loss: 0.767396
(Iteration 3401 / 4900) loss: 0.637031
(Iteration 3421 / 4900) loss: 0.667311
(Iteration 3441 / 4900) loss: 1.015631
(Iteration 3461 / 4900) loss: 0.916772
(Iteration 3481 / 4900) loss: 0.998392
(Iteration 3501 / 4900) loss: 0.757146
(Iteration 3521 / 4900) loss: 0.743694
(Iteration 3541 / 4900) loss: 0.863900
(Iteration 3561 / 4900) loss: 0.917261
(Iteration 3581 / 4900) loss: 0.815331
(Iteration 3601 / 4900) loss: 0.717613
(Iteration 3621 / 4900) loss: 0.652739
(Iteration 3641 / 4900) loss: 0.712831
(Iteration 3661 / 4900) loss: 0.886455
(Iteration 3681 / 4900) loss: 0.798627
(Iteration 3701 / 4900) loss: 0.778502
(Iteration 3721 / 4900) loss: 0.777646
(Iteration 3741 / 4900) loss: 0.917713
(Iteration 3761 / 4900) loss: 0.946531
(Iteration 3781 / 4900) loss: 0.722010
(Iteration 3801 / 4900) loss: 0.797333
(Iteration 3821 / 4900) loss: 0.749392
(Iteration 3841 / 4900) loss: 0.800504
(Iteration 3861 / 4900) loss: 0.871494
(Iteration 3881 / 4900) loss: 0.847518
(Iteration 3901 / 4900) loss: 0.921704
(Epoch 19 / 5) train acc: 0.786000; val_acc: 0.634000
(Iteration 3921 / 4900) loss: 0.781779
(Iteration 3941 / 4900) loss: 0.704124
(Iteration 3961 / 4900) loss: 0.800113
(Iteration 3981 / 4900) loss: 0.958109
(Iteration 4001 / 4900) loss: 0.759886
(Iteration 4021 / 4900) loss: 1.182498
(Iteration 4041 / 4900) loss: 0.672510
(Iteration 4061 / 4900) loss: 0.885097
(Iteration 4081 / 4900) loss: 0.852719
(Iteration 4101 / 4900) loss: 0.757427
(Iteration 4121 / 4900) loss: 0.864267
(Iteration 4141 / 4900) loss: 1.042360
(Iteration 4161 / 4900) loss: 0.661389
(Iteration 4181 / 4900) loss: 0.772770
(Iteration 4201 / 4900) loss: 0.597335
(Iteration 4221 / 4900) loss: 0.832552
(Iteration 4241 / 4900) loss: 0.588319
(Iteration 4261 / 4900) loss: 0.916641
(Iteration 4281 / 4900) loss: 0.855355
(Iteration 4301 / 4900) loss: 0.688942
(Iteration 4321 / 4900) loss: 0.747575
(Iteration 4341 / 4900) loss: 0.749195
(Iteration 4361 / 4900) loss: 0.913384
```

```
(Iteration 4381 / 4900) loss: 0.743450
(Iteration 4401 / 4900) loss: 0.740189
(Iteration 4421 / 4900) loss: 0.620585
(Iteration 4441 / 4900) loss: 0.721129
(Iteration 4461 / 4900) loss: 0.822031
(Iteration 4481 / 4900) loss: 0.675112
(Iteration 4501 / 4900) loss: 0.808639
(Iteration 4521 / 4900) loss: 0.628751
(Iteration 4541 / 4900) loss: 0.722058
(Iteration 4561 / 4900) loss: 0.628966
(Iteration 4581 / 4900) loss: 0.909504
(Iteration 4601 / 4900) loss: 0.597226
(Iteration 4621 / 4900) loss: 1.102409
(Iteration 4641 / 4900) loss: 1.006450
(Iteration 4661 / 4900) loss: 0.856037
(Iteration 4681 / 4900) loss: 1.239151
(Iteration 4701 / 4900) loss: 0.655408
(Iteration 4721 / 4900) loss: 0.642636
(Iteration 4741 / 4900) loss: 1.193529
(Iteration 4761 / 4900) loss: 1.071594
(Iteration 4781 / 4900) loss: 1.199290
(Iteration 4801 / 4900) loss: 0.529035
(Iteration 4821 / 4900) loss: 0.943310
(Iteration 4841 / 4900) loss: 1.343490
(Iteration 4861 / 4900) loss: 0.802157
(Iteration 4881 / 4900) loss: 0.584943
(Epoch 20 / 5) train acc: 0.789000; val_acc: 0.647000
```

In [31]:
```
trained_model_2 = copy.deepcopy(model)
solver.num_epochs = 5
solver.optim_config['learning_rate'] = 1e-5
solver.train()
```

```
(Iteration 1 / 4900) loss: 0.747935
(Epoch 20 / 5) train acc: 0.774000; val_acc: 0.648000
(Iteration 21 / 4900) loss: 1.141088
(Iteration 41 / 4900) loss: 1.003153
(Iteration 61 / 4900) loss: 0.653332
(Iteration 81 / 4900) loss: 0.890407
(Iteration 101 / 4900) loss: 0.857729
(Iteration 121 / 4900) loss: 0.841799
(Iteration 141 / 4900) loss: 0.791224
(Iteration 161 / 4900) loss: 0.875455
(Iteration 181 / 4900) loss: 0.924541
(Iteration 201 / 4900) loss: 0.799955
(Iteration 221 / 4900) loss: 0.760363
(Iteration 241 / 4900) loss: 0.923951
(Iteration 261 / 4900) loss: 0.925741
(Iteration 281 / 4900) loss: 1.018977
(Iteration 301 / 4900) loss: 0.873794
(Iteration 321 / 4900) loss: 0.652875
(Iteration 341 / 4900) loss: 0.769537
(Iteration 361 / 4900) loss: 0.622420
(Iteration 381 / 4900) loss: 0.846045
(Iteration 401 / 4900) loss: 0.855341
(Iteration 421 / 4900) loss: 0.835377
(Iteration 441 / 4900) loss: 0.678336
(Iteration 461 / 4900) loss: 0.735334
(Iteration 481 / 4900) loss: 0.763496
(Iteration 501 / 4900) loss: 0.577646
(Iteration 521 / 4900) loss: 1.031241
(Iteration 541 / 4900) loss: 0.861506
(Iteration 561 / 4900) loss: 0.791081
(Iteration 581 / 4900) loss: 0.701862
(Iteration 601 / 4900) loss: 0.701235
(Iteration 621 / 4900) loss: 0.905675
(Iteration 641 / 4900) loss: 0.957550
(Iteration 661 / 4900) loss: 0.935059
(Iteration 681 / 4900) loss: 0.907963
(Iteration 701 / 4900) loss: 0.809822
(Iteration 721 / 4900) loss: 0.975462
(Iteration 741 / 4900) loss: 0.778250
(Iteration 761 / 4900) loss: 0.969506
(Iteration 781 / 4900) loss: 0.852622
(Iteration 801 / 4900) loss: 0.724149
(Iteration 821 / 4900) loss: 0.892114
(Iteration 841 / 4900) loss: 0.792185
(Iteration 861 / 4900) loss: 0.769015
(Iteration 881 / 4900) loss: 0.843923
(Iteration 901 / 4900) loss: 0.664513
(Iteration 921 / 4900) loss: 0.604002
(Iteration 941 / 4900) loss: 1.119420
(Iteration 961 / 4900) loss: 1.284949
(Epoch 21 / 5) train acc: 0.804000; val_acc: 0.637000
(Iteration 981 / 4900) loss: 0.831183
(Iteration 1001 / 4900) loss: 0.948996
(Iteration 1021 / 4900) loss: 0.791851
(Iteration 1041 / 4900) loss: 0.866627
(Iteration 1061 / 4900) loss: 0.781936
```

```
(Iteration 1081 / 4900) loss: 0.947253
(Iteration 1101 / 4900) loss: 0.869477
(Iteration 1121 / 4900) loss: 0.826357
(Iteration 1141 / 4900) loss: 0.947141
(Iteration 1161 / 4900) loss: 0.953079
(Iteration 1181 / 4900) loss: 0.845107
(Iteration 1201 / 4900) loss: 0.741008
(Iteration 1221 / 4900) loss: 0.991048
(Iteration 1241 / 4900) loss: 0.936330
(Iteration 1261 / 4900) loss: 0.955551
(Iteration 1281 / 4900) loss: 0.778478
(Iteration 1301 / 4900) loss: 0.847852
(Iteration 1321 / 4900) loss: 0.904862
(Iteration 1341 / 4900) loss: 0.741177
(Iteration 1361 / 4900) loss: 1.283020
(Iteration 1381 / 4900) loss: 0.708570
(Iteration 1401 / 4900) loss: 0.719181
(Iteration 1421 / 4900) loss: 0.671178
(Iteration 1441 / 4900) loss: 0.840512
(Iteration 1461 / 4900) loss: 0.730863
(Iteration 1481 / 4900) loss: 0.808435
(Iteration 1501 / 4900) loss: 0.806227
(Iteration 1521 / 4900) loss: 0.683114
(Iteration 1541 / 4900) loss: 0.798101
(Iteration 1561 / 4900) loss: 0.914976
(Iteration 1581 / 4900) loss: 0.748690
(Iteration 1601 / 4900) loss: 0.779880
(Iteration 1621 / 4900) loss: 0.807406
(Iteration 1641 / 4900) loss: 0.644569
(Iteration 1661 / 4900) loss: 0.843262
(Iteration 1681 / 4900) loss: 1.068752
(Iteration 1701 / 4900) loss: 0.849436
(Iteration 1721 / 4900) loss: 1.083786
(Iteration 1741 / 4900) loss: 0.969088
(Iteration 1761 / 4900) loss: 0.734229
(Iteration 1781 / 4900) loss: 0.842071
(Iteration 1801 / 4900) loss: 0.516054
(Iteration 1821 / 4900) loss: 0.923025
(Iteration 1841 / 4900) loss: 0.737720
(Iteration 1861 / 4900) loss: 0.775605
(Iteration 1881 / 4900) loss: 0.874826
(Iteration 1901 / 4900) loss: 0.942776
(Iteration 1921 / 4900) loss: 0.636259
(Iteration 1941 / 4900) loss: 0.670039
(Epoch 22 / 5) train acc: 0.788000; val_acc: 0.635000
(Iteration 1961 / 4900) loss: 0.870204
(Iteration 1981 / 4900) loss: 0.838752
(Iteration 2001 / 4900) loss: 0.775353
(Iteration 2021 / 4900) loss: 0.654184
(Iteration 2041 / 4900) loss: 0.807227
(Iteration 2061 / 4900) loss: 0.729680
(Iteration 2081 / 4900) loss: 0.827167
(Iteration 2101 / 4900) loss: 0.729182
(Iteration 2121 / 4900) loss: 0.537868
(Iteration 2141 / 4900) loss: 0.790328
(Iteration 2161 / 4900) loss: 0.731051
```

```
(Iteration 2181 / 4900) loss: 0.779090
(Iteration 2201 / 4900) loss: 1.011267
(Iteration 2221 / 4900) loss: 0.766413
(Iteration 2241 / 4900) loss: 1.033772
(Iteration 2261 / 4900) loss: 0.792054
(Iteration 2281 / 4900) loss: 0.771625
(Iteration 2301 / 4900) loss: 0.763595
(Iteration 2321 / 4900) loss: 0.799884
(Iteration 2341 / 4900) loss: 0.802646
(Iteration 2361 / 4900) loss: 0.780481
(Iteration 2381 / 4900) loss: 0.589604
(Iteration 2401 / 4900) loss: 0.812422
(Iteration 2421 / 4900) loss: 1.049404
(Iteration 2441 / 4900) loss: 0.779379
(Iteration 2461 / 4900) loss: 0.749898
(Iteration 2481 / 4900) loss: 0.779138
(Iteration 2501 / 4900) loss: 0.717099
(Iteration 2521 / 4900) loss: 0.870276
(Iteration 2541 / 4900) loss: 0.922017
(Iteration 2561 / 4900) loss: 0.829894
(Iteration 2581 / 4900) loss: 0.822408
(Iteration 2601 / 4900) loss: 0.916683
(Iteration 2621 / 4900) loss: 0.840938
(Iteration 2641 / 4900) loss: 0.820376
(Iteration 2661 / 4900) loss: 0.960124
(Iteration 2681 / 4900) loss: 0.787565
(Iteration 2701 / 4900) loss: 0.623074
(Iteration 2721 / 4900) loss: 0.933753
(Iteration 2741 / 4900) loss: 0.937978
(Iteration 2761 / 4900) loss: 0.818481
(Iteration 2781 / 4900) loss: 0.748456
(Iteration 2801 / 4900) loss: 0.715453
(Iteration 2821 / 4900) loss: 1.000924
(Iteration 2841 / 4900) loss: 0.718356
(Iteration 2861 / 4900) loss: 0.696919
(Iteration 2881 / 4900) loss: 0.994122
(Iteration 2901 / 4900) loss: 0.787650
(Iteration 2921 / 4900) loss: 0.822066
(Epoch 23 / 5) train acc: 0.773000; val_acc: 0.623000
(Iteration 2941 / 4900) loss: 1.039891
(Iteration 2961 / 4900) loss: 0.709189
(Iteration 2981 / 4900) loss: 0.615199
(Iteration 3001 / 4900) loss: 0.828792
(Iteration 3021 / 4900) loss: 0.835515
(Iteration 3041 / 4900) loss: 0.775906
(Iteration 3061 / 4900) loss: 0.704985
(Iteration 3081 / 4900) loss: 0.828214
(Iteration 3101 / 4900) loss: 0.874914
(Iteration 3121 / 4900) loss: 0.738135
(Iteration 3141 / 4900) loss: 0.675049
(Iteration 3161 / 4900) loss: 0.955600
(Iteration 3181 / 4900) loss: 0.571925
(Iteration 3201 / 4900) loss: 0.820999
(Iteration 3221 / 4900) loss: 0.717089
(Iteration 3241 / 4900) loss: 0.884665
(Iteration 3261 / 4900) loss: 1.165795
```

```
(Iteration 3281 / 4900) loss: 0.730276
(Iteration 3301 / 4900) loss: 0.773872
(Iteration 3321 / 4900) loss: 0.731832
(Iteration 3341 / 4900) loss: 0.595679
(Iteration 3361 / 4900) loss: 0.690234
(Iteration 3381 / 4900) loss: 0.838822
(Iteration 3401 / 4900) loss: 0.746809
(Iteration 3421 / 4900) loss: 0.770151
(Iteration 3441 / 4900) loss: 0.848355
(Iteration 3461 / 4900) loss: 0.743655
(Iteration 3481 / 4900) loss: 0.687466
(Iteration 3501 / 4900) loss: 0.841512
(Iteration 3521 / 4900) loss: 0.678957
(Iteration 3541 / 4900) loss: 0.800004
(Iteration 3561 / 4900) loss: 0.785756
(Iteration 3581 / 4900) loss: 0.686368
(Iteration 3601 / 4900) loss: 0.899972
(Iteration 3621 / 4900) loss: 0.586928
(Iteration 3641 / 4900) loss: 0.669500
(Iteration 3661 / 4900) loss: 0.872550
(Iteration 3681 / 4900) loss: 0.643720
(Iteration 3701 / 4900) loss: 0.879054
(Iteration 3721 / 4900) loss: 0.724939
(Iteration 3741 / 4900) loss: 0.875207
(Iteration 3761 / 4900) loss: 0.963747
(Iteration 3781 / 4900) loss: 0.919776
(Iteration 3801 / 4900) loss: 0.758989
(Iteration 3821 / 4900) loss: 0.747089
(Iteration 3841 / 4900) loss: 0.681663
(Iteration 3861 / 4900) loss: 0.757273
(Iteration 3881 / 4900) loss: 0.723552
(Iteration 3901 / 4900) loss: 0.901625
(Epoch 24 / 5) train acc: 0.794000; val_acc: 0.622000
(Iteration 3921 / 4900) loss: 0.784694
(Iteration 3941 / 4900) loss: 1.055687
(Iteration 3961 / 4900) loss: 0.606072
(Iteration 3981 / 4900) loss: 0.736625
(Iteration 4001 / 4900) loss: 0.568726
(Iteration 4021 / 4900) loss: 0.766793
(Iteration 4041 / 4900) loss: 0.875969
(Iteration 4061 / 4900) loss: 0.826837
(Iteration 4081 / 4900) loss: 0.778355
(Iteration 4101 / 4900) loss: 0.706743
(Iteration 4121 / 4900) loss: 0.756195
(Iteration 4141 / 4900) loss: 0.902885
(Iteration 4161 / 4900) loss: 0.632632
(Iteration 4181 / 4900) loss: 0.857590
(Iteration 4201 / 4900) loss: 0.832893
(Iteration 4221 / 4900) loss: 0.925008
(Iteration 4241 / 4900) loss: 0.672264
(Iteration 4261 / 4900) loss: 0.850784
(Iteration 4281 / 4900) loss: 0.789245
(Iteration 4301 / 4900) loss: 0.944822
(Iteration 4321 / 4900) loss: 0.720343
(Iteration 4341 / 4900) loss: 1.085008
(Iteration 4361 / 4900) loss: 0.633732
```

```
(Iteration 4381 / 4900) loss: 0.960295
(Iteration 4401 / 4900) loss: 0.975579
(Iteration 4421 / 4900) loss: 0.644086
(Iteration 4441 / 4900) loss: 0.791107
(Iteration 4461 / 4900) loss: 1.202305
(Iteration 4481 / 4900) loss: 1.058798
(Iteration 4501 / 4900) loss: 0.561065
(Iteration 4521 / 4900) loss: 0.660779
(Iteration 4541 / 4900) loss: 0.690544
(Iteration 4561 / 4900) loss: 0.628610
(Iteration 4581 / 4900) loss: 0.787057
(Iteration 4601 / 4900) loss: 0.861475
(Iteration 4621 / 4900) loss: 0.789269
(Iteration 4641 / 4900) loss: 0.735443
(Iteration 4661 / 4900) loss: 0.934900
(Iteration 4681 / 4900) loss: 0.724576
(Iteration 4701 / 4900) loss: 0.844764
(Iteration 4721 / 4900) loss: 0.804635
(Iteration 4741 / 4900) loss: 0.674427
(Iteration 4761 / 4900) loss: 0.675624
(Iteration 4781 / 4900) loss: 0.769665
(Iteration 4801 / 4900) loss: 0.703791
(Iteration 4821 / 4900) loss: 0.889520
(Iteration 4841 / 4900) loss: 0.905426
(Iteration 4861 / 4900) loss: 0.694405
(Iteration 4881 / 4900) loss: 0.955674
(Epoch 25 / 5) train acc: 0.798000; val_acc: 0.604000
```

In [32]:
```
trained_model_3 = copy.deepcopy(model)
solver.num_epochs = 5
solver.optim_config['learning_rate'] = 1e-5
solver.train()
```

```
(Iteration 1 / 4900) loss: 0.704525
(Epoch 25 / 5) train acc: 0.810000; val_acc: 0.646000
(Iteration 21 / 4900) loss: 0.900114
(Iteration 41 / 4900) loss: 1.009866
(Iteration 61 / 4900) loss: 0.906801
(Iteration 81 / 4900) loss: 0.901012
(Iteration 101 / 4900) loss: 0.986584
(Iteration 121 / 4900) loss: 0.741993
(Iteration 141 / 4900) loss: 0.750318
(Iteration 161 / 4900) loss: 1.127374
(Iteration 181 / 4900) loss: 0.877690
(Iteration 201 / 4900) loss: 0.760076
(Iteration 221 / 4900) loss: 0.694472
(Iteration 241 / 4900) loss: 0.774694
(Iteration 261 / 4900) loss: 0.996501
(Iteration 281 / 4900) loss: 0.937348
(Iteration 301 / 4900) loss: 0.796746
(Iteration 321 / 4900) loss: 0.983020
(Iteration 341 / 4900) loss: 0.806626
(Iteration 361 / 4900) loss: 0.990587
(Iteration 381 / 4900) loss: 0.901725
(Iteration 401 / 4900) loss: 0.769700
(Iteration 421 / 4900) loss: 0.832428
(Iteration 441 / 4900) loss: 0.818562
(Iteration 461 / 4900) loss: 0.857432
(Iteration 481 / 4900) loss: 0.619886
(Iteration 501 / 4900) loss: 0.749346
(Iteration 521 / 4900) loss: 1.079864
(Iteration 541 / 4900) loss: 0.470067
(Iteration 561 / 4900) loss: 0.723277
(Iteration 581 / 4900) loss: 0.955766
(Iteration 601 / 4900) loss: 0.887281
(Iteration 621 / 4900) loss: 0.737453
(Iteration 641 / 4900) loss: 0.690975
(Iteration 661 / 4900) loss: 0.861040
(Iteration 681 / 4900) loss: 0.918809
(Iteration 701 / 4900) loss: 0.907756
(Iteration 721 / 4900) loss: 0.849807
(Iteration 741 / 4900) loss: 0.748536
(Iteration 761 / 4900) loss: 0.741511
(Iteration 781 / 4900) loss: 0.646307
(Iteration 801 / 4900) loss: 0.700000
(Iteration 821 / 4900) loss: 0.975844
(Iteration 841 / 4900) loss: 1.154302
(Iteration 861 / 4900) loss: 0.871302
(Iteration 881 / 4900) loss: 0.917474
(Iteration 901 / 4900) loss: 0.957913
(Iteration 921 / 4900) loss: 0.760178
(Iteration 941 / 4900) loss: 0.772178
(Iteration 961 / 4900) loss: 0.883553
(Epoch 26 / 5) train acc: 0.773000; val_acc: 0.626000
(Iteration 981 / 4900) loss: 0.649714
(Iteration 1001 / 4900) loss: 0.965794
(Iteration 1021 / 4900) loss: 0.745430
(Iteration 1041 / 4900) loss: 0.811781
(Iteration 1061 / 4900) loss: 1.012289
```

```
(Iteration 1081 / 4900) loss: 0.765191
(Iteration 1101 / 4900) loss: 0.742750
(Iteration 1121 / 4900) loss: 0.679885
(Iteration 1141 / 4900) loss: 0.728944
(Iteration 1161 / 4900) loss: 0.700147
(Iteration 1181 / 4900) loss: 0.924633
(Iteration 1201 / 4900) loss: 0.736811
(Iteration 1221 / 4900) loss: 0.750885
(Iteration 1241 / 4900) loss: 0.887327
(Iteration 1261 / 4900) loss: 1.285640
(Iteration 1281 / 4900) loss: 0.699865
(Iteration 1301 / 4900) loss: 0.480105
(Iteration 1321 / 4900) loss: 0.915138
(Iteration 1341 / 4900) loss: 0.772750
(Iteration 1361 / 4900) loss: 0.777782
(Iteration 1381 / 4900) loss: 1.028827
(Iteration 1401 / 4900) loss: 0.670049
(Iteration 1421 / 4900) loss: 0.926546
(Iteration 1441 / 4900) loss: 0.785927
(Iteration 1461 / 4900) loss: 0.767774
(Iteration 1481 / 4900) loss: 1.051205
(Iteration 1501 / 4900) loss: 0.738980
(Iteration 1521 / 4900) loss: 0.844757
(Iteration 1541 / 4900) loss: 0.604799
(Iteration 1561 / 4900) loss: 0.906581
(Iteration 1581 / 4900) loss: 1.097028
(Iteration 1601 / 4900) loss: 0.902030
(Iteration 1621 / 4900) loss: 0.908357
(Iteration 1641 / 4900) loss: 0.707549
(Iteration 1661 / 4900) loss: 0.738320
(Iteration 1681 / 4900) loss: 0.743362
(Iteration 1701 / 4900) loss: 0.643714
(Iteration 1721 / 4900) loss: 0.863860
(Iteration 1741 / 4900) loss: 0.871333
(Iteration 1761 / 4900) loss: 0.887483
(Iteration 1781 / 4900) loss: 0.815713
(Iteration 1801 / 4900) loss: 0.772572
(Iteration 1821 / 4900) loss: 0.834030
(Iteration 1841 / 4900) loss: 0.876622
(Iteration 1861 / 4900) loss: 1.080448
(Iteration 1881 / 4900) loss: 0.795460
(Iteration 1901 / 4900) loss: 0.587335
(Iteration 1921 / 4900) loss: 0.890564
(Iteration 1941 / 4900) loss: 0.966020
(Epoch 27 / 5) train acc: 0.804000; val_acc: 0.626000
(Iteration 1961 / 4900) loss: 0.808488
(Iteration 1981 / 4900) loss: 0.918186
(Iteration 2001 / 4900) loss: 0.646845
(Iteration 2021 / 4900) loss: 1.014897
(Iteration 2041 / 4900) loss: 0.999694
(Iteration 2061 / 4900) loss: 0.750478
(Iteration 2081 / 4900) loss: 0.789316
(Iteration 2101 / 4900) loss: 0.733147
(Iteration 2121 / 4900) loss: 1.167443
(Iteration 2141 / 4900) loss: 0.847986
(Iteration 2161 / 4900) loss: 0.704283
```

```
(Iteration 2181 / 4900) loss: 0.787521
(Iteration 2201 / 4900) loss: 1.038674
(Iteration 2221 / 4900) loss: 0.861221
(Iteration 2241 / 4900) loss: 1.151764
(Iteration 2261 / 4900) loss: 0.803214
(Iteration 2281 / 4900) loss: 0.793223
(Iteration 2301 / 4900) loss: 0.899599
(Iteration 2321 / 4900) loss: 0.612996
(Iteration 2341 / 4900) loss: 0.947242
(Iteration 2361 / 4900) loss: 0.791511
(Iteration 2381 / 4900) loss: 0.985661
(Iteration 2401 / 4900) loss: 0.754657
(Iteration 2421 / 4900) loss: 0.866225
(Iteration 2441 / 4900) loss: 0.661216
(Iteration 2461 / 4900) loss: 0.716147
(Iteration 2481 / 4900) loss: 0.738953
(Iteration 2501 / 4900) loss: 0.858524
(Iteration 2521 / 4900) loss: 0.668482
(Iteration 2541 / 4900) loss: 0.712783
(Iteration 2561 / 4900) loss: 1.341448
(Iteration 2581 / 4900) loss: 0.846584
(Iteration 2601 / 4900) loss: 0.726183
(Iteration 2621 / 4900) loss: 0.862796
(Iteration 2641 / 4900) loss: 0.779188
(Iteration 2661 / 4900) loss: 0.947631
(Iteration 2681 / 4900) loss: 0.654696
(Iteration 2701 / 4900) loss: 0.718987
(Iteration 2721 / 4900) loss: 0.844954
(Iteration 2741 / 4900) loss: 0.693450
(Iteration 2761 / 4900) loss: 0.975416
(Iteration 2781 / 4900) loss: 0.732124
(Iteration 2801 / 4900) loss: 1.085350
(Iteration 2821 / 4900) loss: 0.889319
(Iteration 2841 / 4900) loss: 0.769723
(Iteration 2861 / 4900) loss: 0.779792
(Iteration 2881 / 4900) loss: 0.701170
(Iteration 2901 / 4900) loss: 0.746616
(Iteration 2921 / 4900) loss: 0.791179
(Epoch 28 / 5) train acc: 0.767000; val_acc: 0.601000
(Iteration 2941 / 4900) loss: 1.132664
(Iteration 2961 / 4900) loss: 0.861565
(Iteration 2981 / 4900) loss: 0.855819
(Iteration 3001 / 4900) loss: 0.843696
(Iteration 3021 / 4900) loss: 0.773703
(Iteration 3041 / 4900) loss: 0.618213
(Iteration 3061 / 4900) loss: 0.659597
(Iteration 3081 / 4900) loss: 1.096191
(Iteration 3101 / 4900) loss: 1.046600
(Iteration 3121 / 4900) loss: 0.966551
(Iteration 3141 / 4900) loss: 0.800777
(Iteration 3161 / 4900) loss: 0.489993
(Iteration 3181 / 4900) loss: 0.721466
(Iteration 3201 / 4900) loss: 1.074788
(Iteration 3221 / 4900) loss: 0.638793
(Iteration 3241 / 4900) loss: 0.954447
(Iteration 3261 / 4900) loss: 0.654144
```

```
(Iteration 3281 / 4900) loss: 0.688029
(Iteration 3301 / 4900) loss: 0.938373
(Iteration 3321 / 4900) loss: 0.797835
(Iteration 3341 / 4900) loss: 0.927348
(Iteration 3361 / 4900) loss: 1.087167
(Iteration 3381 / 4900) loss: 0.664515
(Iteration 3401 / 4900) loss: 0.824904
(Iteration 3421 / 4900) loss: 0.968775
(Iteration 3441 / 4900) loss: 0.739172
(Iteration 3461 / 4900) loss: 1.011698
(Iteration 3481 / 4900) loss: 0.678263
(Iteration 3501 / 4900) loss: 1.149382
(Iteration 3521 / 4900) loss: 0.684981
(Iteration 3541 / 4900) loss: 0.742008
(Iteration 3561 / 4900) loss: 0.699364
(Iteration 3581 / 4900) loss: 0.964464
(Iteration 3601 / 4900) loss: 0.885898
(Iteration 3621 / 4900) loss: 0.926855
(Iteration 3641 / 4900) loss: 0.872959
(Iteration 3661 / 4900) loss: 0.806146
(Iteration 3681 / 4900) loss: 1.014210
(Iteration 3701 / 4900) loss: 0.803981
(Iteration 3721 / 4900) loss: 0.794233
(Iteration 3741 / 4900) loss: 0.887958
(Iteration 3761 / 4900) loss: 1.175933
(Iteration 3781 / 4900) loss: 0.691703
(Iteration 3801 / 4900) loss: 0.685932
(Iteration 3821 / 4900) loss: 1.235406
(Iteration 3841 / 4900) loss: 0.760938
(Iteration 3861 / 4900) loss: 0.697897
(Iteration 3881 / 4900) loss: 0.534026
(Iteration 3901 / 4900) loss: 0.656776
(Epoch 29 / 5) train acc: 0.783000; val_acc: 0.646000
(Iteration 3921 / 4900) loss: 0.696758
(Iteration 3941 / 4900) loss: 1.028426
(Iteration 3961 / 4900) loss: 0.775927
(Iteration 3981 / 4900) loss: 0.925633
(Iteration 4001 / 4900) loss: 0.763476
(Iteration 4021 / 4900) loss: 0.670076
(Iteration 4041 / 4900) loss: 0.973747
(Iteration 4061 / 4900) loss: 1.010335
(Iteration 4081 / 4900) loss: 0.871942
(Iteration 4101 / 4900) loss: 0.708696
(Iteration 4121 / 4900) loss: 0.815095
(Iteration 4141 / 4900) loss: 0.652432
(Iteration 4161 / 4900) loss: 0.834603
(Iteration 4181 / 4900) loss: 0.776959
(Iteration 4201 / 4900) loss: 0.719387
(Iteration 4221 / 4900) loss: 0.952218
(Iteration 4241 / 4900) loss: 0.844544
(Iteration 4261 / 4900) loss: 0.875285
(Iteration 4281 / 4900) loss: 0.767946
(Iteration 4301 / 4900) loss: 0.739311
(Iteration 4321 / 4900) loss: 0.627611
(Iteration 4341 / 4900) loss: 0.986803
(Iteration 4361 / 4900) loss: 0.985489
```

```
(Iteration 4381 / 4900) loss: 0.825656
(Iteration 4401 / 4900) loss: 0.576003
(Iteration 4421 / 4900) loss: 0.795369
(Iteration 4441 / 4900) loss: 0.990532
(Iteration 4461 / 4900) loss: 0.748590
(Iteration 4481 / 4900) loss: 1.026869
(Iteration 4501 / 4900) loss: 0.975630
(Iteration 4521 / 4900) loss: 0.636898
(Iteration 4541 / 4900) loss: 0.810363
(Iteration 4561 / 4900) loss: 0.776447
(Iteration 4581 / 4900) loss: 0.803301
(Iteration 4601 / 4900) loss: 0.751270
(Iteration 4621 / 4900) loss: 0.732213
(Iteration 4641 / 4900) loss: 0.689238
(Iteration 4661 / 4900) loss: 0.909167
(Iteration 4681 / 4900) loss: 0.841916
(Iteration 4701 / 4900) loss: 0.735680
(Iteration 4721 / 4900) loss: 0.791213
(Iteration 4741 / 4900) loss: 0.870773
(Iteration 4761 / 4900) loss: 0.913647
(Iteration 4781 / 4900) loss: 0.954115
(Iteration 4801 / 4900) loss: 0.686273
(Iteration 4821 / 4900) loss: 0.645491
(Iteration 4841 / 4900) loss: 0.829937
(Iteration 4861 / 4900) loss: 0.704715
(Iteration 4881 / 4900) loss: 0.714759
(Epoch 30 / 5) train acc: 0.784000; val_acc: 0.615000
```

In [36]:
```
trained_model_4 = copy.deepcopy(model)
```

In [37]:
```
model = ConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=16,
                      filter_size=3)

solver = Solver(model, data,
                num_epochs=30, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=100)
solver.train()
trained_model_5 = copy.deepcopy(model)
```

```
(Iteration 1 / 29400) loss: 2.302793
(Epoch 0 / 30) train acc: 0.075000; val_acc: 0.090000
(Iteration 101 / 29400) loss: 1.691937
(Iteration 201 / 29400) loss: 1.650450
(Iteration 301 / 29400) loss: 1.859370
(Iteration 401 / 29400) loss: 1.648756
(Iteration 501 / 29400) loss: 1.507317
(Iteration 601 / 29400) loss: 1.324527
(Iteration 701 / 29400) loss: 1.232898
(Iteration 801 / 29400) loss: 1.229140
(Iteration 901 / 29400) loss: 1.557349
(Epoch 1 / 30) train acc: 0.524000; val_acc: 0.519000
(Iteration 1001 / 29400) loss: 0.981531
(Iteration 1101 / 29400) loss: 1.403056
(Iteration 1201 / 29400) loss: 1.171621
(Iteration 1301 / 29400) loss: 1.308847
(Iteration 1401 / 29400) loss: 1.467566
(Iteration 1501 / 29400) loss: 1.442070
(Iteration 1601 / 29400) loss: 1.141891
(Iteration 1701 / 29400) loss: 1.023134
(Iteration 1801 / 29400) loss: 1.067709
(Iteration 1901 / 29400) loss: 1.034076
(Epoch 2 / 30) train acc: 0.602000; val_acc: 0.555000
(Iteration 2001 / 29400) loss: 1.223675
(Iteration 2101 / 29400) loss: 1.255370
(Iteration 2201 / 29400) loss: 1.248882
(Iteration 2301 / 29400) loss: 1.395759
(Iteration 2401 / 29400) loss: 0.983885
(Iteration 2501 / 29400) loss: 1.008691
(Iteration 2601 / 29400) loss: 1.117892
(Iteration 2701 / 29400) loss: 1.260315
(Iteration 2801 / 29400) loss: 1.017411
(Iteration 2901 / 29400) loss: 0.888043
(Epoch 3 / 30) train acc: 0.642000; val_acc: 0.599000
(Iteration 3001 / 29400) loss: 1.093375
(Iteration 3101 / 29400) loss: 0.882365
(Iteration 3201 / 29400) loss: 1.150990
(Iteration 3301 / 29400) loss: 1.053396
(Iteration 3401 / 29400) loss: 1.155522
(Iteration 3501 / 29400) loss: 0.730538
(Iteration 3601 / 29400) loss: 1.366116
(Iteration 3701 / 29400) loss: 1.150994
(Iteration 3801 / 29400) loss: 1.058955
(Iteration 3901 / 29400) loss: 1.182111
(Epoch 4 / 30) train acc: 0.678000; val_acc: 0.634000
(Iteration 4001 / 29400) loss: 1.136660
(Iteration 4101 / 29400) loss: 1.020015
(Iteration 4201 / 29400) loss: 1.048076
(Iteration 4301 / 29400) loss: 1.008837
(Iteration 4401 / 29400) loss: 1.098965
(Iteration 4501 / 29400) loss: 1.044809
(Iteration 4601 / 29400) loss: 0.754198
(Iteration 4701 / 29400) loss: 1.091259
(Iteration 4801 / 29400) loss: 1.040217
(Epoch 5 / 30) train acc: 0.710000; val_acc: 0.624000
(Iteration 4901 / 29400) loss: 1.121863
```

```
(Iteration 5001 / 29400) loss: 1.289684
(Iteration 5101 / 29400) loss: 1.003106
(Iteration 5201 / 29400) loss: 1.049670
(Iteration 5301 / 29400) loss: 0.780800
(Iteration 5401 / 29400) loss: 0.764046
(Iteration 5501 / 29400) loss: 1.065119
(Iteration 5601 / 29400) loss: 1.159663
(Iteration 5701 / 29400) loss: 1.110991
(Iteration 5801 / 29400) loss: 1.156640
(Epoch 6 / 30) train acc: 0.698000; val_acc: 0.606000
(Iteration 5901 / 29400) loss: 1.305939
(Iteration 6001 / 29400) loss: 1.025397
(Iteration 6101 / 29400) loss: 0.802389
(Iteration 6201 / 29400) loss: 0.876753
(Iteration 6301 / 29400) loss: 0.803745
(Iteration 6401 / 29400) loss: 1.068041
(Iteration 6501 / 29400) loss: 1.030384
(Iteration 6601 / 29400) loss: 1.040704
(Iteration 6701 / 29400) loss: 1.016260
(Iteration 6801 / 29400) loss: 0.854872
(Epoch 7 / 30) train acc: 0.705000; val_acc: 0.593000
(Iteration 6901 / 29400) loss: 1.105781
(Iteration 7001 / 29400) loss: 1.041199
(Iteration 7101 / 29400) loss: 0.924834
(Iteration 7201 / 29400) loss: 0.744242
(Iteration 7301 / 29400) loss: 0.865305
(Iteration 7401 / 29400) loss: 0.779924
(Iteration 7501 / 29400) loss: 0.892988
(Iteration 7601 / 29400) loss: 0.948969
(Iteration 7701 / 29400) loss: 0.852140
(Iteration 7801 / 29400) loss: 0.838735
(Epoch 8 / 30) train acc: 0.741000; val_acc: 0.632000
(Iteration 7901 / 29400) loss: 1.096632
(Iteration 8001 / 29400) loss: 1.022911
(Iteration 8101 / 29400) loss: 0.838377
(Iteration 8201 / 29400) loss: 1.004262
(Iteration 8301 / 29400) loss: 1.284084
(Iteration 8401 / 29400) loss: 0.763525
(Iteration 8501 / 29400) loss: 1.287397
(Iteration 8601 / 29400) loss: 0.961609
(Iteration 8701 / 29400) loss: 1.225301
(Iteration 8801 / 29400) loss: 0.798924
(Epoch 9 / 30) train acc: 0.743000; val_acc: 0.608000
(Iteration 8901 / 29400) loss: 0.926263
(Iteration 9001 / 29400) loss: 1.032054
(Iteration 9101 / 29400) loss: 0.775054
(Iteration 9201 / 29400) loss: 1.028804
(Iteration 9301 / 29400) loss: 1.187614
(Iteration 9401 / 29400) loss: 0.925146
(Iteration 9501 / 29400) loss: 0.962608
(Iteration 9601 / 29400) loss: 0.693171
(Iteration 9701 / 29400) loss: 1.040684
(Epoch 10 / 30) train acc: 0.743000; val_acc: 0.596000
(Iteration 9801 / 29400) loss: 1.144903
(Iteration 9901 / 29400) loss: 0.830952
(Iteration 10001 / 29400) loss: 0.923964
```

```
(Iteration 10101 / 29400) loss: 0.901824
(Iteration 10201 / 29400) loss: 0.985760
(Iteration 10301 / 29400) loss: 0.846967
(Iteration 10401 / 29400) loss: 0.832074
(Iteration 10501 / 29400) loss: 0.785540
(Iteration 10601 / 29400) loss: 0.931452
(Iteration 10701 / 29400) loss: 0.638206
(Epoch 11 / 30) train acc: 0.746000; val_acc: 0.628000
(Iteration 10801 / 29400) loss: 0.954991
(Iteration 10901 / 29400) loss: 0.926483
(Iteration 11001 / 29400) loss: 1.024129
(Iteration 11101 / 29400) loss: 0.809300
(Iteration 11201 / 29400) loss: 0.932281
(Iteration 11301 / 29400) loss: 0.780059
(Iteration 11401 / 29400) loss: 1.105221
(Iteration 11501 / 29400) loss: 0.889036
(Iteration 11601 / 29400) loss: 0.750047
(Iteration 11701 / 29400) loss: 0.839989
(Epoch 12 / 30) train acc: 0.749000; val_acc: 0.630000
(Iteration 11801 / 29400) loss: 0.929033
(Iteration 11901 / 29400) loss: 0.731614
(Iteration 12001 / 29400) loss: 0.825958
(Iteration 12101 / 29400) loss: 0.810733
(Iteration 12201 / 29400) loss: 0.816651
(Iteration 12301 / 29400) loss: 0.660336
(Iteration 12401 / 29400) loss: 1.039457
(Iteration 12501 / 29400) loss: 0.944234
(Iteration 12601 / 29400) loss: 0.748803
(Iteration 12701 / 29400) loss: 1.001065
(Epoch 13 / 30) train acc: 0.770000; val_acc: 0.624000
(Iteration 12801 / 29400) loss: 0.984778
(Iteration 12901 / 29400) loss: 0.924951
(Iteration 13001 / 29400) loss: 1.119974
(Iteration 13101 / 29400) loss: 0.686494
(Iteration 13201 / 29400) loss: 0.728504
(Iteration 13301 / 29400) loss: 0.853088
(Iteration 13401 / 29400) loss: 0.868575
(Iteration 13501 / 29400) loss: 1.203691
(Iteration 13601 / 29400) loss: 0.825615
(Iteration 13701 / 29400) loss: 0.850250
(Epoch 14 / 30) train acc: 0.784000; val_acc: 0.626000
(Iteration 13801 / 29400) loss: 0.639687
(Iteration 13901 / 29400) loss: 1.017595
(Iteration 14001 / 29400) loss: 0.784880
(Iteration 14101 / 29400) loss: 0.899356
(Iteration 14201 / 29400) loss: 0.905502
(Iteration 14301 / 29400) loss: 0.706228
(Iteration 14401 / 29400) loss: 0.937561
(Iteration 14501 / 29400) loss: 0.957987
(Iteration 14601 / 29400) loss: 0.936106
(Epoch 15 / 30) train acc: 0.755000; val_acc: 0.593000
(Iteration 14701 / 29400) loss: 0.771479
(Iteration 14801 / 29400) loss: 0.951993
(Iteration 14901 / 29400) loss: 0.928432
(Iteration 15001 / 29400) loss: 0.768112
(Iteration 15101 / 29400) loss: 0.816051
```

```
(Iteration 15201 / 29400) loss: 0.758214
(Iteration 15301 / 29400) loss: 1.095046
(Iteration 15401 / 29400) loss: 0.681201
(Iteration 15501 / 29400) loss: 0.813693
(Iteration 15601 / 29400) loss: 1.083790
(Epoch 16 / 30) train acc: 0.815000; val_acc: 0.608000
(Iteration 15701 / 29400) loss: 0.848852
(Iteration 15801 / 29400) loss: 0.836384
(Iteration 15901 / 29400) loss: 0.799547
(Iteration 16001 / 29400) loss: 0.877625
(Iteration 16101 / 29400) loss: 0.733230
(Iteration 16201 / 29400) loss: 0.835206
(Iteration 16301 / 29400) loss: 0.994498
(Iteration 16401 / 29400) loss: 0.715817
(Iteration 16501 / 29400) loss: 0.911459
(Iteration 16601 / 29400) loss: 0.769343
(Epoch 17 / 30) train acc: 0.761000; val_acc: 0.609000
(Iteration 16701 / 29400) loss: 0.873516
(Iteration 16801 / 29400) loss: 0.980463
(Iteration 16901 / 29400) loss: 0.804346
(Iteration 17001 / 29400) loss: 0.835737
(Iteration 17101 / 29400) loss: 0.940823
(Iteration 17201 / 29400) loss: 0.799333
(Iteration 17301 / 29400) loss: 0.750838
(Iteration 17401 / 29400) loss: 1.267343
(Iteration 17501 / 29400) loss: 0.993972
(Iteration 17601 / 29400) loss: 0.746152
(Epoch 18 / 30) train acc: 0.793000; val_acc: 0.608000
(Iteration 17701 / 29400) loss: 0.679050
(Iteration 17801 / 29400) loss: 0.979362
(Iteration 17901 / 29400) loss: 0.908690
(Iteration 18001 / 29400) loss: 0.767674
(Iteration 18101 / 29400) loss: 0.723963
(Iteration 18201 / 29400) loss: 0.673012
(Iteration 18301 / 29400) loss: 1.023257
(Iteration 18401 / 29400) loss: 0.732338
(Iteration 18501 / 29400) loss: 0.645888
(Iteration 18601 / 29400) loss: 0.821157
(Epoch 19 / 30) train acc: 0.797000; val_acc: 0.596000
(Iteration 18701 / 29400) loss: 0.796700
(Iteration 18801 / 29400) loss: 0.955525
(Iteration 18901 / 29400) loss: 0.662875
(Iteration 19001 / 29400) loss: 0.688047
(Iteration 19101 / 29400) loss: 0.861145
(Iteration 19201 / 29400) loss: 0.849388
(Iteration 19301 / 29400) loss: 0.953776
(Iteration 19401 / 29400) loss: 0.768129
(Iteration 19501 / 29400) loss: 0.723347
(Epoch 20 / 30) train acc: 0.790000; val_acc: 0.618000
(Iteration 19601 / 29400) loss: 1.055478
(Iteration 19701 / 29400) loss: 0.661921
(Iteration 19801 / 29400) loss: 0.893799
(Iteration 19901 / 29400) loss: 0.910891
(Iteration 20001 / 29400) loss: 0.859217
(Iteration 20101 / 29400) loss: 0.955965
(Iteration 20201 / 29400) loss: 0.837237
```

```
(Iteration 20301 / 29400) loss: 0.826297
(Iteration 20401 / 29400) loss: 0.781760
(Iteration 20501 / 29400) loss: 0.782307
(Epoch 21 / 30) train acc: 0.779000; val_acc: 0.612000
(Iteration 20601 / 29400) loss: 0.797338
(Iteration 20701 / 29400) loss: 0.833131
(Iteration 20801 / 29400) loss: 0.771707
(Iteration 20901 / 29400) loss: 0.871094
(Iteration 21001 / 29400) loss: 0.670081
(Iteration 21101 / 29400) loss: 0.837427
(Iteration 21201 / 29400) loss: 0.982284
(Iteration 21301 / 29400) loss: 1.004983
(Iteration 21401 / 29400) loss: 0.679258
(Iteration 21501 / 29400) loss: 1.032522
(Epoch 22 / 30) train acc: 0.792000; val_acc: 0.613000
(Iteration 21601 / 29400) loss: 0.943139
(Iteration 21701 / 29400) loss: 0.658635
(Iteration 21801 / 29400) loss: 0.557717
(Iteration 21901 / 29400) loss: 0.729655
(Iteration 22001 / 29400) loss: 0.552028
(Iteration 22101 / 29400) loss: 0.930180
(Iteration 22201 / 29400) loss: 0.894013
(Iteration 22301 / 29400) loss: 0.838214
(Iteration 22401 / 29400) loss: 0.786227
(Iteration 22501 / 29400) loss: 0.947070
(Epoch 23 / 30) train acc: 0.818000; val_acc: 0.621000
(Iteration 22601 / 29400) loss: 0.775884
(Iteration 22701 / 29400) loss: 0.623547
(Iteration 22801 / 29400) loss: 0.710013
(Iteration 22901 / 29400) loss: 0.684451
(Iteration 23001 / 29400) loss: 0.794045
(Iteration 23101 / 29400) loss: 0.751583
(Iteration 23201 / 29400) loss: 0.775912
(Iteration 23301 / 29400) loss: 0.946679
(Iteration 23401 / 29400) loss: 0.657180
(Iteration 23501 / 29400) loss: 0.867496
(Epoch 24 / 30) train acc: 0.785000; val_acc: 0.603000
(Iteration 23601 / 29400) loss: 0.804917
(Iteration 23701 / 29400) loss: 1.016203
(Iteration 23801 / 29400) loss: 0.728799
(Iteration 23901 / 29400) loss: 0.756580
(Iteration 24001 / 29400) loss: 0.962883
(Iteration 24101 / 29400) loss: 0.870621
(Iteration 24201 / 29400) loss: 0.675973
(Iteration 24301 / 29400) loss: 0.974950
(Iteration 24401 / 29400) loss: 0.671275
(Epoch 25 / 30) train acc: 0.798000; val_acc: 0.621000
(Iteration 24501 / 29400) loss: 0.712933
(Iteration 24601 / 29400) loss: 0.558919
(Iteration 24701 / 29400) loss: 0.750502
(Iteration 24801 / 29400) loss: 0.573233
(Iteration 24901 / 29400) loss: 0.649213
(Iteration 25001 / 29400) loss: 0.716516
(Iteration 25101 / 29400) loss: 0.505424
(Iteration 25201 / 29400) loss: 0.784607
(Iteration 25301 / 29400) loss: 0.889633
```

```
(Iteration 25401 / 29400) loss: 0.588985
(Epoch 26 / 30) train acc: 0.835000; val_acc: 0.617000
(Iteration 25501 / 29400) loss: 0.663351
(Iteration 25601 / 29400) loss: 0.744674
(Iteration 25701 / 29400) loss: 0.990664
(Iteration 25801 / 29400) loss: 0.561355
(Iteration 25901 / 29400) loss: 0.650748
(Iteration 26001 / 29400) loss: 0.955350
(Iteration 26101 / 29400) loss: 0.688709
(Iteration 26201 / 29400) loss: 0.823475
(Iteration 26301 / 29400) loss: 0.615981
(Iteration 26401 / 29400) loss: 1.370893
(Epoch 27 / 30) train acc: 0.797000; val_acc: 0.594000
(Iteration 26501 / 29400) loss: 0.706660
(Iteration 26601 / 29400) loss: 0.784579
(Iteration 26701 / 29400) loss: 0.496402
(Iteration 26801 / 29400) loss: 0.891004
(Iteration 26901 / 29400) loss: 0.891403
(Iteration 27001 / 29400) loss: 0.932165
(Iteration 27101 / 29400) loss: 0.690677
(Iteration 27201 / 29400) loss: 0.699764
(Iteration 27301 / 29400) loss: 0.797611
(Iteration 27401 / 29400) loss: 0.688552
(Epoch 28 / 30) train acc: 0.816000; val_acc: 0.588000
(Iteration 27501 / 29400) loss: 0.873456
(Iteration 27601 / 29400) loss: 0.808285
(Iteration 27701 / 29400) loss: 0.616474
(Iteration 27801 / 29400) loss: 0.670000
(Iteration 27901 / 29400) loss: 0.708996
(Iteration 28001 / 29400) loss: 0.839293
(Iteration 28101 / 29400) loss: 0.642384
(Iteration 28201 / 29400) loss: 0.677442
(Iteration 28301 / 29400) loss: 0.942901
(Iteration 28401 / 29400) loss: 0.608838
(Epoch 29 / 30) train acc: 0.776000; val_acc: 0.594000
(Iteration 28501 / 29400) loss: 0.729514
(Iteration 28601 / 29400) loss: 1.314294
(Iteration 28701 / 29400) loss: 0.676368
(Iteration 28801 / 29400) loss: 0.689135
(Iteration 28901 / 29400) loss: 0.955335
(Iteration 29001 / 29400) loss: 0.951013
(Iteration 29101 / 29400) loss: 0.672456
(Iteration 29201 / 29400) loss: 0.645167
(Iteration 29301 / 29400) loss: 0.963412
(Epoch 30 / 30) train acc: 0.805000; val_acc: 0.589000
```

In [55]:
```
model = trained_model_3
solver.num_epochs = 5
solver.optim_config['learning_rate'] = 1e-5
solver.train()
trained_model_6 = copy.deepcopy(model)
```

```
(Iteration 1 / 4900) loss: 0.929024
(Epoch 30 / 5) train acc: 0.693000; val_acc: 0.649000
(Iteration 101 / 4900) loss: 1.154466
(Iteration 201 / 4900) loss: 1.082536
(Iteration 301 / 4900) loss: 1.239102
(Iteration 401 / 4900) loss: 0.883638
(Iteration 501 / 4900) loss: 0.877873
(Iteration 601 / 4900) loss: 0.954301
(Iteration 701 / 4900) loss: 1.014482
(Iteration 801 / 4900) loss: 1.306716
(Iteration 901 / 4900) loss: 0.843590
(Epoch 31 / 5) train acc: 0.687000; val_acc: 0.588000
(Iteration 1001 / 4900) loss: 1.057001
(Iteration 1101 / 4900) loss: 1.128325
(Iteration 1201 / 4900) loss: 1.093451
(Iteration 1301 / 4900) loss: 1.090247
(Iteration 1401 / 4900) loss: 1.154203
(Iteration 1501 / 4900) loss: 0.930361
(Iteration 1601 / 4900) loss: 0.746443
(Iteration 1701 / 4900) loss: 1.140484
(Iteration 1801 / 4900) loss: 0.932391
(Iteration 1901 / 4900) loss: 0.778857
(Epoch 32 / 5) train acc: 0.709000; val_acc: 0.610000
(Iteration 2001 / 4900) loss: 0.670056
(Iteration 2101 / 4900) loss: 0.991045
(Iteration 2201 / 4900) loss: 1.515977
(Iteration 2301 / 4900) loss: 0.772692
(Iteration 2401 / 4900) loss: 0.803600
(Iteration 2501 / 4900) loss: 0.827669
(Iteration 2601 / 4900) loss: 0.839351
(Iteration 2701 / 4900) loss: 0.974267
(Iteration 2801 / 4900) loss: 0.964205
(Iteration 2901 / 4900) loss: 0.925325
(Epoch 33 / 5) train acc: 0.715000; val_acc: 0.640000
(Iteration 3001 / 4900) loss: 0.798051
(Iteration 3101 / 4900) loss: 1.009907
(Iteration 3201 / 4900) loss: 0.994866
(Iteration 3301 / 4900) loss: 0.850496
(Iteration 3401 / 4900) loss: 1.158252
(Iteration 3501 / 4900) loss: 1.176737
(Iteration 3601 / 4900) loss: 1.015842
(Iteration 3701 / 4900) loss: 1.068358
(Iteration 3801 / 4900) loss: 1.074806
(Iteration 3901 / 4900) loss: 1.002657
(Epoch 34 / 5) train acc: 0.708000; val_acc: 0.627000
(Iteration 4001 / 4900) loss: 0.926338
(Iteration 4101 / 4900) loss: 0.647171
(Iteration 4201 / 4900) loss: 1.010344
(Iteration 4301 / 4900) loss: 0.629603
(Iteration 4401 / 4900) loss: 0.889685
(Iteration 4501 / 4900) loss: 0.931304
(Iteration 4601 / 4900) loss: 0.951754
(Iteration 4701 / 4900) loss: 0.695574
(Iteration 4801 / 4900) loss: 0.850618
(Epoch 35 / 5) train acc: 0.725000; val_acc: 0.618000
```

In [84]:
```
average_model = ConvNet(weight_scale=0.001, hidden_dim=50, reg=0.002, num_filters=1
                        filter_size=3)

# Averaging best models -  This does not give best validation acc
average_model.params['W1'] =  (trained_model_2.params['W1'] + trained_model_3.param
average_model.params['b1'] =  (trained_model_2.params['b1'] + trained_model_3.param
average_model.params['W2'] =  (trained_model_2.params['W2'] + trained_model_3.param
average_model.params['b2'] =  (trained_model_2.params['b2'] + trained_model_3.param
average_model.params['W3'] =  (trained_model_2.params['W3'] + trained_model_3.param
average_model.params['b3'] =  (trained_model_2.params['b3'] + trained_model_3.param
average_model.params['W4'] =  (trained_model_2.params['W4'] + trained_model_3.param
average_model.params['b4'] =  (trained_model_2.params['b4'] + trained_model_3.param
```

Model 3 gives best validation accuracy

In [87]:
```
model = trained_model_3
```

Show Your Accuracies

In [88]:
```
sample_train_sz = 1000
y_train_hat = np.argmax(model.loss(data['X_train'][:sample_train_sz]), axis=1)
print ('Training set accuracy: ', np.sum(y_train_hat == data['y_train'][:sample_tra

y_val_hat = np.argmax(model.loss(data['X_val']), axis=1)
print ('Validation set accuracy: ', np.sum(y_val_hat == data['y_val'])*100/float(da

y_test_hat = np.argmax(model.loss(data['X_test']), axis=1)
print ('Test set accuracy: ', np.sum(y_test_hat == data['y_test'])*100/float(data['
```

```
Training set accuracy:  78.2 %
Validation set accuracy:  64.8 %
Test set accuracy:  61.6 %
```

Extra Credit Description

If you implement any additional features for extra credit, clearly describe them here with pointers to any code in this or other files if applicable.

1. Added the convnet.py class.

- Added a conv_relu_forward layer to the ConvNet. (location: cs6353/classifiers/convnet.py)

2. Experimented with several hyper-parameter settings.
3. Experimented with adaptive learning rate instead of a static learning rate.
4. Experimented with averaging the best models and using the averaged model for prediction. Although this did not give the best results.