FullyConnectedNets

December 12, 2024

1 Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

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

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

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

  return out, cache
```

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

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

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

```
  return dx, dw
```

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

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

[24]:
```
# 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. 'cs6353/assignments/assignment3/'
FOLDERNAME = 'cs6353/assignments/assignment3/'
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

# Install requirements from colab_requirements.txt
# TODO: Please change your path below to the colab_requirements.txt file
! python -m pip install -r /content/drive/My\ Drive/$FOLDERNAME/requirements.txt
```

```
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/assignment3/cs6353/datasets
--2024-12-12 21:30:09--  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…
connected.
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  31.6MB/s    in 5.3s

2024-12-12 21:30:15 (30.9 MB/s) - ‘cifar-10-python.tar.gz’ saved
```

```
[170498071/170498071]

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/assignment3
Collecting attrs==19.1.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 1))
  Using cached attrs-19.1.0-py2.py3-none-any.whl.metadata (10 kB)
Collecting backcall==0.1.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 2))
  Using cached backcall-0.1.0.zip (11 kB)
  Preparing metadata (setup.py) … done
Collecting bleach==3.1.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 3))
  Using cached bleach-3.1.0-py2.py3-none-any.whl.metadata (19 kB)
Collecting certifi==2019.6.16 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 4))
  Using cached certifi-2019.6.16-py2.py3-none-any.whl.metadata (2.5 kB)
Collecting cycler==0.10.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 5))
  Using cached cycler-0.10.0-py2.py3-none-any.whl.metadata (722 bytes)
Collecting decorator==4.4.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 6))
  Using cached decorator-4.4.0-py2.py3-none-any.whl.metadata (3.7 kB)
Collecting defusedxml==0.6.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 7))
  Using cached defusedxml-0.6.0-py2.py3-none-any.whl.metadata (31 kB)
Collecting entrypoints==0.3 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 8))
  Using cached entrypoints-0.3-py2.py3-none-any.whl.metadata (1.4 kB)
Collecting future==0.17.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 9))
  Using cached future-0.17.1.tar.gz (829 kB)
  Preparing metadata (setup.py) … done
Collecting imageio==2.5.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 10))
  Using cached imageio-2.5.0-py3-none-any.whl.metadata (2.8 kB)
Collecting ipykernel==5.1.2 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 11))
  Using cached ipykernel-5.1.2-py3-none-any.whl.metadata (919 bytes)
Collecting ipython==7.8.0 (from -r /content/drive/My
```

```
Drive/cs6353/assignments/assignment3//requirements.txt (line 12))
  Using cached ipython-7.8.0-py3-none-any.whl.metadata (4.3 kB)
Requirement already satisfied: ipython-genutils==0.2.0 in
/usr/local/lib/python3.10/dist-packages (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 13)) (0.2.0)
Collecting ipywidgets==7.5.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 14))
  Using cached ipywidgets-7.5.1-py2.py3-none-any.whl.metadata (1.8 kB)
Collecting jedi==0.15.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 15))
  Using cached jedi-0.15.1-py2.py3-none-any.whl.metadata (15 kB)
Collecting Jinja2==2.10.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 16))
  Using cached Jinja2-2.10.1-py2.py3-none-any.whl.metadata (2.2 kB)
Collecting jsonschema==3.0.2 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 17))
  Using cached jsonschema-3.0.2-py2.py3-none-any.whl.metadata (7.4 kB)
Collecting jupyter==1.0.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 18))
  Using cached jupyter-1.0.0-py2.py3-none-any.whl.metadata (995 bytes)
Collecting jupyter-client==5.3.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 19))
  Using cached jupyter_client-5.3.1-py2.py3-none-any.whl.metadata (3.6 kB)
Collecting jupyter-console==6.0.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 20))
  Using cached jupyter_console-6.0.0-py2.py3-none-any.whl.metadata (955 bytes)
Collecting jupyter-core==4.5.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 21))
  Using cached jupyter_core-4.5.0-py2.py3-none-any.whl.metadata (884 bytes)
Collecting kiwisolver==1.1.0 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 22))
  Using cached kiwisolver-1.1.0.tar.gz (30 kB)
  Preparing metadata (setup.py) … done
Collecting MarkupSafe==1.1.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 23))
  Using cached MarkupSafe-1.1.1.tar.gz (19 kB)
  Preparing metadata (setup.py) … done
Collecting matplotlib==3.1.1 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 24))
  Using cached matplotlib-3.1.1.tar.gz (37.8 MB)
  Preparing metadata (setup.py) … done
Collecting mistune==0.8.4 (from -r /content/drive/My
Drive/cs6353/assignments/assignment3//requirements.txt (line 25))
  Using cached mistune-0.8.4-py2.py3-none-any.whl.metadata (8.5 kB)
```

```
ERROR: Could not find a version that satisfies the requirement mkl-
fft==1.0.6 (from versions: 1.3.6, 1.3.8, 1.3.11)
ERROR: No matching distribution found for mkl-fft==1.0.6
```

[ ]:
```
# 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_array
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))))
```

[ ]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

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

2 Affine layer: foward

Open the file `cs6353/layers.py` and implement the `affine_forward` function.

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

[ ]:
```
# Test the affine_forward function

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

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

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

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

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

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

3 Affine layer: backward

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

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

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

_, cache = affine_forward(x, w, b)
```

```
dx, dw, db = affine_backward(dout, cache)

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

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

4 ReLU activation: forward

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

[ ]:
```
# Test the relu_forward function

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

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

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

5 ReLU activation: backward

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

[ ]:
```
#Test for relu_backward

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

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

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

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

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

5.1 Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour? 1. Sigmoid 2. ReLU 3. Leaky ReLU

5.2 Answer:

1. Sigmold: can have issues with vanishing gradients. In the Sigmol function the function compresses the input range in the range (0,1). As a result when the imput becames very large and positive (x» 0) or very large anad negative (x « 0).The function saturates, causing its gradiant to approach zero. This saturation will sifnificatly slowdown the learning.
    - Problematic Input Areas:
        - Large positive inputs (x » 0): Sigmoid output is close to 1, gradient is near zero.
        - Large negative inputs (x « 0): Sigmoid output is close to 0, gradient is near zero.
2. ReLu:

- ReLu will only have issues in certain specific areas. ReLu uses 0 for all negative inputs.Thus, when the inpiut is negative, the gradient is also zero, which stops gradient flow for those neurouns. Which brings us to the ReLu problem where neurons die and stop updating because they always output 0.
- Problematic Input Areas:
    - Negative inputs (x $<$ 0): The gradient is exactly zero.

3. Leaky ReLU:

- Leaky ReLu will mitigate the the Relu problem by llowing a small, nonzero gradient for negative inputs by scaling them with a small factor (= 0 01). This make sure that neurons continue to update, even when the input is negative.

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy,

we define several convenience layers in the file `cs6353/layer_utils.py`.

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

[ ]:
```
from cs6353.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

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

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

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

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs6353/layers.py`.

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

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

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

```
# Test svm_loss function. Loss should be around 9 and dx error should be around␣
 ↪the order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

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

# Test softmax_loss function. Loss should be close to 2.3 and dx error should␣
 ↪be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

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

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

8 Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs6353/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

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

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

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
```

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
 ↪33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
 ↪49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
 ↪66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

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

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization …
```

```
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Open the file `cs6353/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

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

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################

from cs6353.data_utils import get_CIFAR10_data
from cs6353.classifiers.fc_net import TwoLayerNet
from cs6353.solver import Solver

# Load and preprocess CIFAR-10 data
data = get_CIFAR10_data()
X_train = data['X_train']
y_train = data['y_train']
X_val = data['X_val']
y_val = data['y_val']

mean_image = np.mean(X_train, axis=0)
X_train -= mean_image
X_val -= mean_image

# Hyperparameters
input_dim = 3 * 32 * 32
```

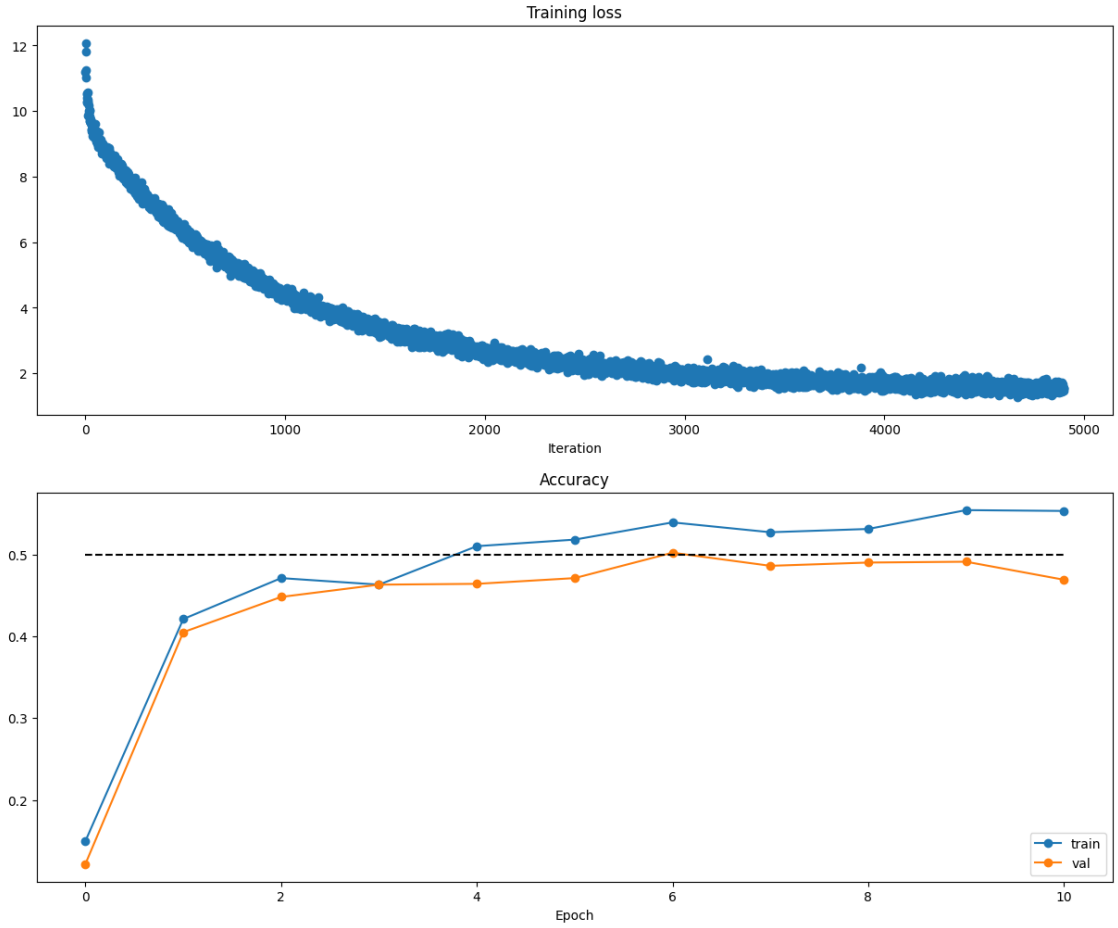
```
hidden_dim = 100
num_classes = 10
weight_scale = 1e-2
reg = 0.5

# Create TwoLayerNet instance
model = TwoLayerNet(input_dim=input_dim, hidden_dim=hidden_dim,␣
 ↪num_classes=num_classes,
                    weight_scale=weight_scale, reg=reg)

# Data dictionary
data = {
    'X_train': X_train,
    'y_train': y_train,
    'X_val': X_val,
    'y_val': y_val,
}

# Solver configuration
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                    'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=10,
                batch_size=100,
                print_every=100)

# Train the model
solver.train()

# Print the best validation accuracy
print('Best validation accuracy: ', solver.best_val_acc)

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
(Iteration 1 / 4900) loss: 11.186109
(Epoch 0 / 10) train acc: 0.150000; val_acc: 0.122000
(Iteration 101 / 4900) loss: 8.704228
(Iteration 201 / 4900) loss: 8.198292
(Iteration 301 / 4900) loss: 7.308988
(Iteration 401 / 4900) loss: 6.633375
(Epoch 1 / 10) train acc: 0.421000; val_acc: 0.405000
(Iteration 501 / 4900) loss: 6.292015
(Iteration 601 / 4900) loss: 5.679113
```

```
(Iteration 701 / 4900) loss: 5.446687
(Iteration 801 / 4900) loss: 5.131491
(Iteration 901 / 4900) loss: 4.623973
(Epoch 2 / 10) train acc: 0.471000; val_acc: 0.448000
(Iteration 1001 / 4900) loss: 4.426431
(Iteration 1101 / 4900) loss: 4.143573
(Iteration 1201 / 4900) loss: 3.839472
(Iteration 1301 / 4900) loss: 3.592786
(Iteration 1401 / 4900) loss: 3.543734
(Epoch 3 / 10) train acc: 0.463000; val_acc: 0.463000
(Iteration 1501 / 4900) loss: 3.336353
(Iteration 1601 / 4900) loss: 3.162963
(Iteration 1701 / 4900) loss: 3.010993
(Iteration 1801 / 4900) loss: 3.047657
(Iteration 1901 / 4900) loss: 2.809534
(Epoch 4 / 10) train acc: 0.510000; val_acc: 0.464000
(Iteration 2001 / 4900) loss: 2.761864
(Iteration 2101 / 4900) loss: 2.649672
(Iteration 2201 / 4900) loss: 2.636208
(Iteration 2301 / 4900) loss: 2.257218
(Iteration 2401 / 4900) loss: 2.237168
(Epoch 5 / 10) train acc: 0.518000; val_acc: 0.471000
(Iteration 2501 / 4900) loss: 2.234852
(Iteration 2601 / 4900) loss: 2.150715
(Iteration 2701 / 4900) loss: 2.203075
(Iteration 2801 / 4900) loss: 2.180923
(Iteration 2901 / 4900) loss: 2.023988
(Epoch 6 / 10) train acc: 0.539000; val_acc: 0.502000
(Iteration 3001 / 4900) loss: 1.916947
(Iteration 3101 / 4900) loss: 1.718164
(Iteration 3201 / 4900) loss: 2.175791
(Iteration 3301 / 4900) loss: 1.804439
(Iteration 3401 / 4900) loss: 1.970542
(Epoch 7 / 10) train acc: 0.527000; val_acc: 0.486000
(Iteration 3501 / 4900) loss: 1.667801
(Iteration 3601 / 4900) loss: 1.594947
(Iteration 3701 / 4900) loss: 1.773476
(Iteration 3801 / 4900) loss: 1.626285
(Iteration 3901 / 4900) loss: 1.455549
(Epoch 8 / 10) train acc: 0.531000; val_acc: 0.490000
(Iteration 4001 / 4900) loss: 1.520912
(Iteration 4101 / 4900) loss: 1.685998
(Iteration 4201 / 4900) loss: 1.543523
(Iteration 4301 / 4900) loss: 1.438639
(Iteration 4401 / 4900) loss: 1.952501
(Epoch 9 / 10) train acc: 0.554000; val_acc: 0.491000
(Iteration 4501 / 4900) loss: 1.563292
(Iteration 4601 / 4900) loss: 1.690163
```

```
(Iteration 4701 / 4900) loss: 1.733680
(Iteration 4801 / 4900) loss: 1.492639
(Epoch 10 / 10) train acc: 0.553000; val_acc: 0.469000
Best validation accuracy:  0.502
```

[ ]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



10 Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs6353/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing batch/layer normalization; we will add those features soon.

10.1 Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

[ ]:
```
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 reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

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

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  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])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
```

```
Initial loss:  7.052114776533016
W1 relative error: 3.90e-09
W2 relative error: 6.87e-08
W3 relative error: 2.13e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.57e-10
```

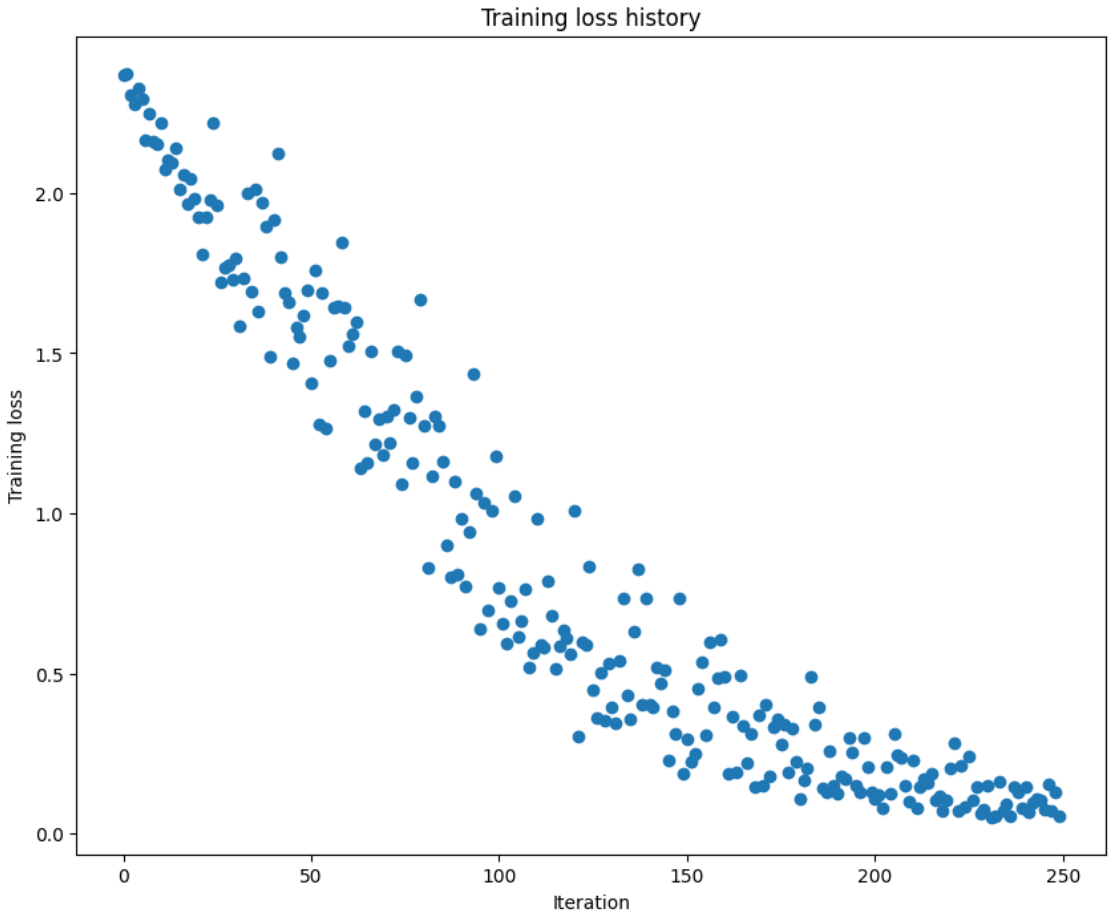
As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

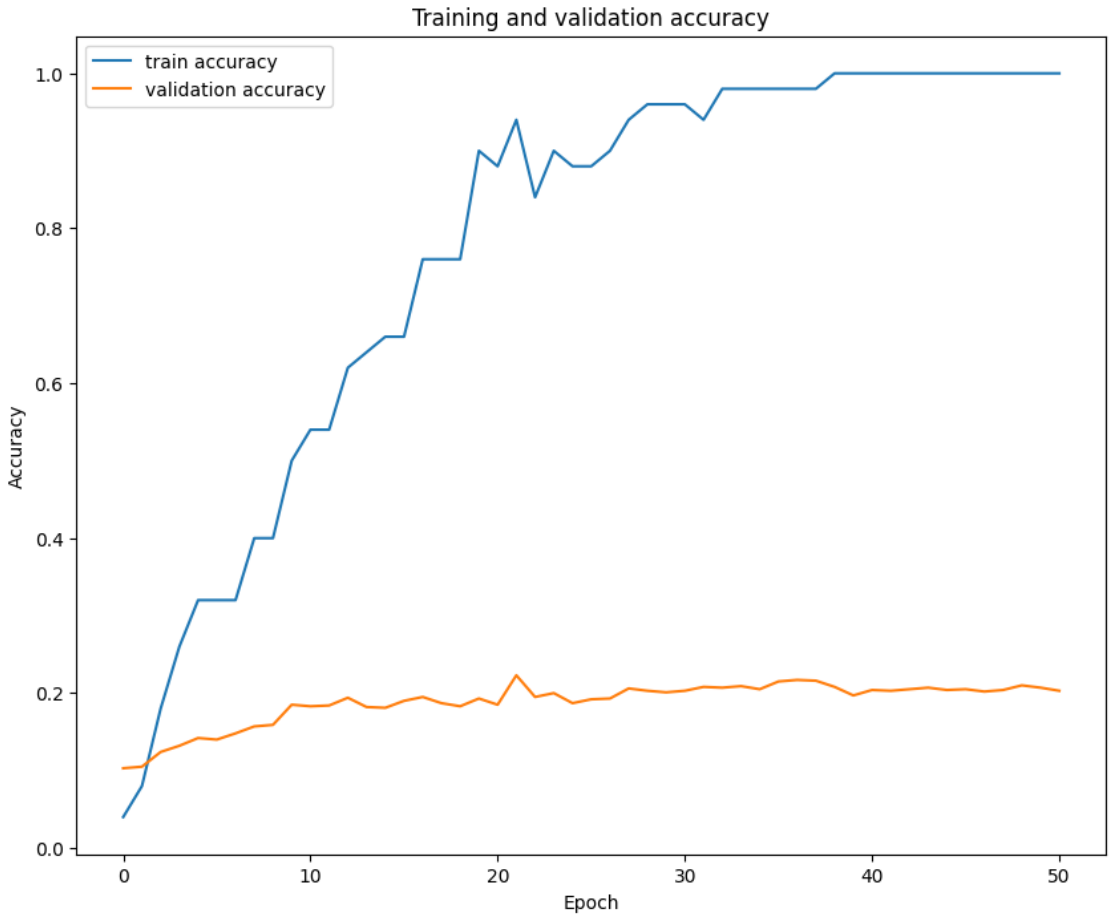
[ ]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train].flatten(),
    'X_val': data['X_val'],
    'y_val': data['y_val'].flatten(),
}

learning_rate = 1e-3  # Start with a moderate learning rate
weight_scale = 1e-2   # Conservative weight initialization
reg = 0.0             # Disable regularization for overfitting

# Define the network
model = FullyConnectedNet([100, 100],
                          weight_scale=weight_scale, reg=reg, dtype=np.float64)

# Use Solver to train on the small dataset
solver = Solver(model, small_data,
                print_every=10, num_epochs=50, batch_size=10,
                update_rule='sgd',
                optim_config={
                    'learning_rate': learning_rate,
                })

# Train the model
solver.train()

# Plot the loss history
plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
# Plot accuracy history
plt.plot(solver.train_acc_history, label='train accuracy')
plt.plot(solver.val_acc_history, label='validation accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
(Iteration 1 / 250) loss: 2.366164
(Epoch 0 / 50) train acc: 0.040000; val_acc: 0.103000
(Epoch 1 / 50) train acc: 0.080000; val_acc: 0.105000
(Epoch 2 / 50) train acc: 0.180000; val_acc: 0.124000
(Iteration 11 / 250) loss: 2.219767
(Epoch 3 / 50) train acc: 0.260000; val_acc: 0.132000
(Epoch 4 / 50) train acc: 0.320000; val_acc: 0.142000
(Iteration 21 / 250) loss: 1.923238
(Epoch 5 / 50) train acc: 0.320000; val_acc: 0.140000
(Epoch 6 / 50) train acc: 0.320000; val_acc: 0.148000
(Iteration 31 / 250) loss: 1.795226
(Epoch 7 / 50) train acc: 0.400000; val_acc: 0.157000
(Epoch 8 / 50) train acc: 0.400000; val_acc: 0.159000
(Iteration 41 / 250) loss: 1.916851
(Epoch 9 / 50) train acc: 0.500000; val_acc: 0.185000
(Epoch 10 / 50) train acc: 0.540000; val_acc: 0.183000
(Iteration 51 / 250) loss: 1.404552
(Epoch 11 / 50) train acc: 0.540000; val_acc: 0.184000
(Epoch 12 / 50) train acc: 0.620000; val_acc: 0.194000
(Iteration 61 / 250) loss: 1.522926
(Epoch 13 / 50) train acc: 0.640000; val_acc: 0.182000
(Epoch 14 / 50) train acc: 0.660000; val_acc: 0.181000
(Iteration 71 / 250) loss: 1.302427
(Epoch 15 / 50) train acc: 0.660000; val_acc: 0.190000
(Epoch 16 / 50) train acc: 0.760000; val_acc: 0.195000
(Iteration 81 / 250) loss: 1.272422
(Epoch 17 / 50) train acc: 0.760000; val_acc: 0.187000
(Epoch 18 / 50) train acc: 0.760000; val_acc: 0.183000
(Iteration 91 / 250) loss: 0.984227
(Epoch 19 / 50) train acc: 0.900000; val_acc: 0.193000
(Epoch 20 / 50) train acc: 0.880000; val_acc: 0.185000
(Iteration 101 / 250) loss: 0.768675
(Epoch 21 / 50) train acc: 0.940000; val_acc: 0.223000
(Epoch 22 / 50) train acc: 0.840000; val_acc: 0.195000
(Iteration 111 / 250) loss: 0.982320
(Epoch 23 / 50) train acc: 0.900000; val_acc: 0.200000
(Epoch 24 / 50) train acc: 0.880000; val_acc: 0.187000
(Iteration 121 / 250) loss: 1.007583
```

```
(Epoch 25 / 50) train acc: 0.880000; val_acc: 0.192000
(Epoch 26 / 50) train acc: 0.900000; val_acc: 0.193000
(Iteration 131 / 250) loss: 0.394380
(Epoch 27 / 50) train acc: 0.940000; val_acc: 0.206000
(Epoch 28 / 50) train acc: 0.960000; val_acc: 0.203000
(Iteration 141 / 250) loss: 0.402931
(Epoch 29 / 50) train acc: 0.960000; val_acc: 0.201000
(Epoch 30 / 50) train acc: 0.960000; val_acc: 0.203000
(Iteration 151 / 250) loss: 0.295562
(Epoch 31 / 50) train acc: 0.940000; val_acc: 0.208000
(Epoch 32 / 50) train acc: 0.980000; val_acc: 0.207000
(Iteration 161 / 250) loss: 0.488932
(Epoch 33 / 50) train acc: 0.980000; val_acc: 0.209000
(Epoch 34 / 50) train acc: 0.980000; val_acc: 0.205000
(Iteration 171 / 250) loss: 0.151954
(Epoch 35 / 50) train acc: 0.980000; val_acc: 0.215000
(Epoch 36 / 50) train acc: 0.980000; val_acc: 0.217000
(Iteration 181 / 250) loss: 0.106743
(Epoch 37 / 50) train acc: 0.980000; val_acc: 0.216000
(Epoch 38 / 50) train acc: 1.000000; val_acc: 0.208000
(Iteration 191 / 250) loss: 0.124291
(Epoch 39 / 50) train acc: 1.000000; val_acc: 0.197000
(Epoch 40 / 50) train acc: 1.000000; val_acc: 0.204000
(Iteration 201 / 250) loss: 0.110018
(Epoch 41 / 50) train acc: 1.000000; val_acc: 0.203000
(Epoch 42 / 50) train acc: 1.000000; val_acc: 0.205000
(Iteration 211 / 250) loss: 0.230059
(Epoch 43 / 50) train acc: 1.000000; val_acc: 0.207000
(Epoch 44 / 50) train acc: 1.000000; val_acc: 0.204000
(Iteration 221 / 250) loss: 0.202486
(Epoch 45 / 50) train acc: 1.000000; val_acc: 0.205000
(Epoch 46 / 50) train acc: 1.000000; val_acc: 0.202000
(Iteration 231 / 250) loss: 0.150378
(Epoch 47 / 50) train acc: 1.000000; val_acc: 0.204000
(Epoch 48 / 50) train acc: 1.000000; val_acc: 0.210000
(Iteration 241 / 250) loss: 0.146843
(Epoch 49 / 50) train acc: 1.000000; val_acc: 0.207000
(Epoch 50 / 50) train acc: 1.000000; val_acc: 0.203000
```


Training loss history
Training loss
Iteration
0.0
0.5
1.0
1.5
2.0
0
50
100
150
200
250



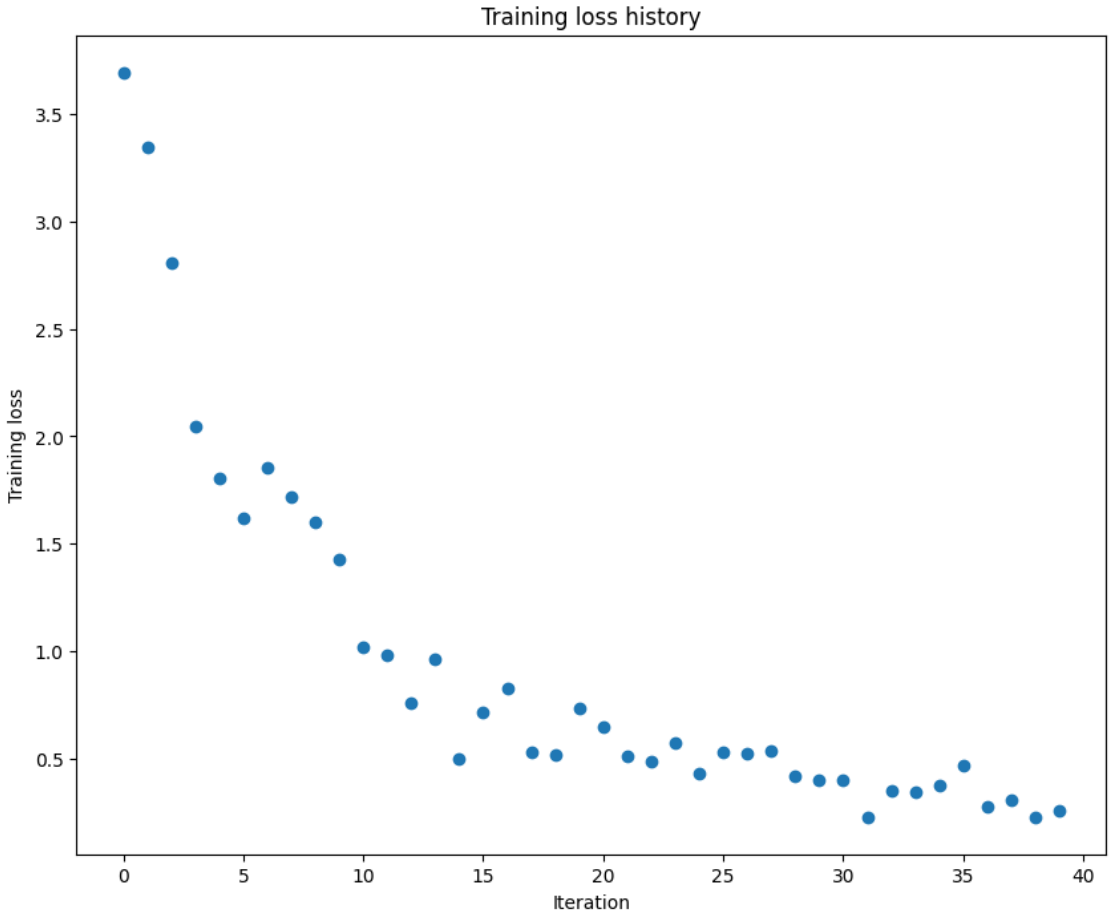
Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

[ ]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

# Adjust learning rate and weight scale
learning_rate = 2e-3  # Start with a smaller learning rate to stabilize training
```

```
weight_scale = 5e-2  # Use a smaller weight initialization scale to avoid␣
 ↪exploding activations

# Define the five-layer network
model = FullyConnectedNet([100, 100, 100, 100],
                          weight_scale=weight_scale, dtype=np.float64)

# Use Solver to train the model on the small dataset
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                    'learning_rate': learning_rate,
                })

# Train the model
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 3.695081
(Epoch 0 / 20) train acc: 0.140000; val_acc: 0.110000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.088000
(Epoch 2 / 20) train acc: 0.380000; val_acc: 0.123000
(Epoch 3 / 20) train acc: 0.480000; val_acc: 0.128000
(Epoch 4 / 20) train acc: 0.560000; val_acc: 0.114000
(Epoch 5 / 20) train acc: 0.640000; val_acc: 0.124000
(Iteration 11 / 40) loss: 1.019097
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.124000
(Epoch 7 / 20) train acc: 0.840000; val_acc: 0.139000
(Epoch 8 / 20) train acc: 0.800000; val_acc: 0.135000
(Epoch 9 / 20) train acc: 0.860000; val_acc: 0.135000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.133000
(Iteration 21 / 40) loss: 0.649122
(Epoch 11 / 20) train acc: 0.920000; val_acc: 0.135000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.120000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.130000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.126000
(Epoch 15 / 20) train acc: 0.940000; val_acc: 0.135000
(Iteration 31 / 40) loss: 0.398907
(Epoch 16 / 20) train acc: 0.960000; val_acc: 0.141000
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.139000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.134000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.138000
```

```
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.134000
```



10.2 Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

10.3 Answer:

1. Sensitivity to Initialization Scale:

A five-layer network is significantly more sensitive to the initialization scale than the three-layer network. Because as we add more layers, the information (like gradients and activations) needs to flow through a deeper stack of layers. If the weights are not initialized properly:

- Too small weights: Activations and gradients shrink layer by layer, leading to vanishing gradients, where the network struggles to learn. This results in very slow or stagnant training.
- In contrast, a three-layer network has fewer layers, so it's less prone to these issues. The gradients don't need to pass through as many layers, making the network more robust to a

wider range of initialization scales.

2. Why Deeper Networks Are More Sensitive:

- Deep networks involve more layers of non-linear transformations. Each layer amplifies or diminishes the activations and gradients depending on the weight initialization. Small issues in initialization compound as they propagate through the network. With a shallow network (like three layers), this compounding effect is much smaller, so it's easier to find a workable initialization scale.
- My Experience:
  - I did in fact experienced that tweaking the initialization scale for the three-layer net only required minor adjustments before it started learning effectively. But for the five-layer net, even small changes to the initialization scale made a big difference in whether the network trained successfully or not. Because in deeper networks smaller problems in the first layer can get magnified by the time they reach the last layer.
- My Takeaway

  Deeper the network, more careful we need to be with initialization to ensure stable learning. Techniques like He initialization or Xavier initialization help, but it becomes increasingly important as we add more layers.

11 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

12 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent.

Open the file `cs6353/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

[ ]:
```
from cs6353.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
```

```
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

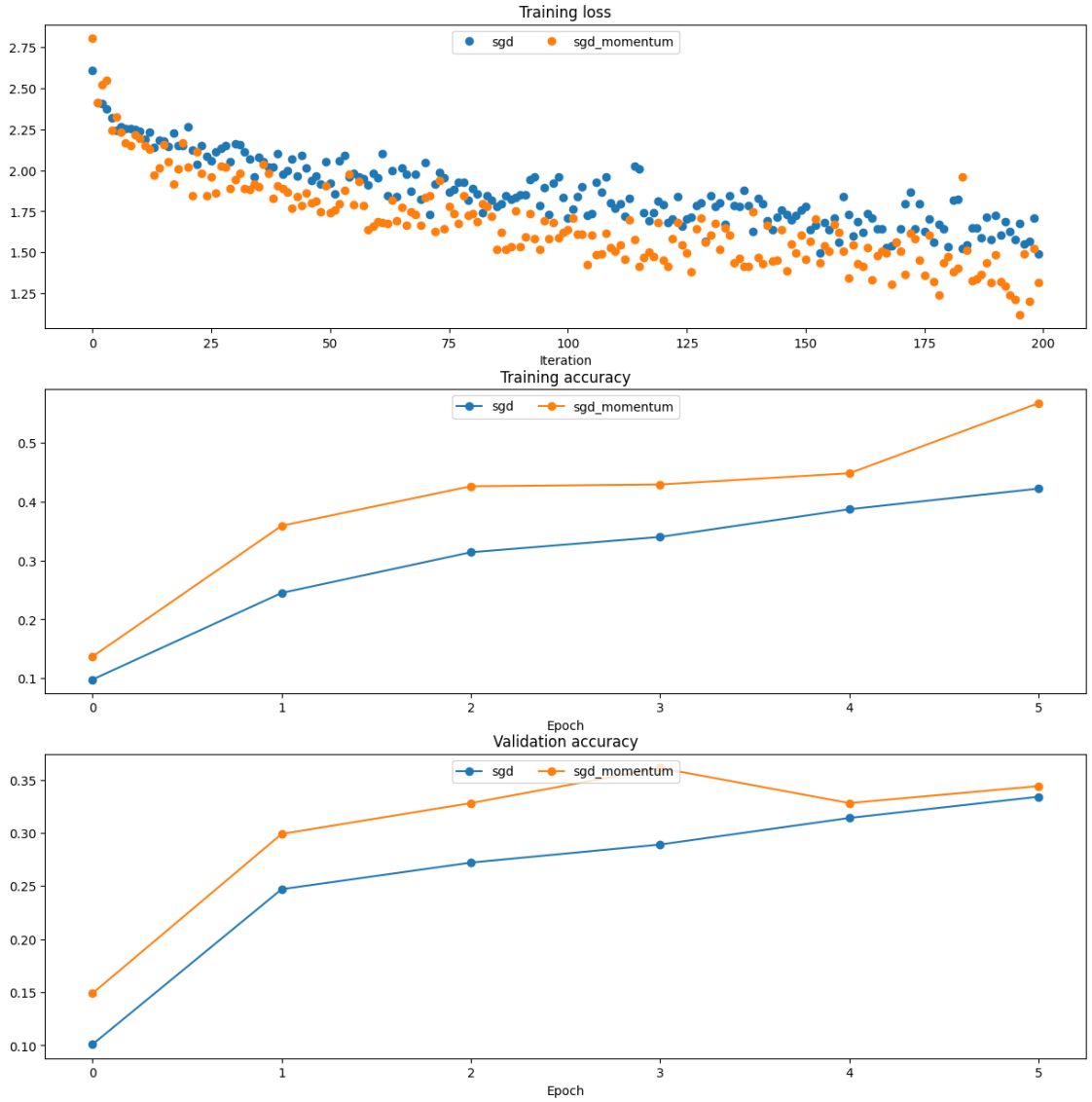
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

[ ]:
```
num_train = 4000
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 = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')
```

```
plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.605014
(Epoch 0 / 5) train acc: 0.098000; val_acc: 0.101000
(Iteration 11 / 200) loss: 2.236982
(Iteration 21 / 200) loss: 2.264074
(Iteration 31 / 200) loss: 2.161986
(Epoch 1 / 5) train acc: 0.245000; val_acc: 0.247000
(Iteration 41 / 200) loss: 1.975239
(Iteration 51 / 200) loss: 1.921147
(Iteration 61 / 200) loss: 1.951534
(Iteration 71 / 200) loss: 2.044217
(Epoch 2 / 5) train acc: 0.314000; val_acc: 0.272000
(Iteration 81 / 200) loss: 1.890290
(Iteration 91 / 200) loss: 1.848685
(Iteration 101 / 200) loss: 1.709628
(Iteration 111 / 200) loss: 1.771198
(Epoch 3 / 5) train acc: 0.340000; val_acc: 0.289000
(Iteration 121 / 200) loss: 1.789582
(Iteration 131 / 200) loss: 1.843255
(Iteration 141 / 200) loss: 1.829962
(Iteration 151 / 200) loss: 1.778582
(Epoch 4 / 5) train acc: 0.387000; val_acc: 0.314000
(Iteration 161 / 200) loss: 1.598128
```

```
(Iteration 171 / 200) loss: 1.644447
(Iteration 181 / 200) loss: 1.535790
(Iteration 191 / 200) loss: 1.725762
(Epoch 5 / 5) train acc: 0.422000; val_acc: 0.334000

running with  sgd_momentum
(Iteration 1 / 200) loss: 2.802887
(Epoch 0 / 5) train acc: 0.137000; val_acc: 0.149000
(Iteration 11 / 200) loss: 2.195209
(Iteration 21 / 200) loss: 2.021857
(Iteration 31 / 200) loss: 1.943642
(Epoch 1 / 5) train acc: 0.359000; val_acc: 0.299000
(Iteration 41 / 200) loss: 1.888625
(Iteration 51 / 200) loss: 1.740094
(Iteration 61 / 200) loss: 1.685759
(Iteration 71 / 200) loss: 1.835642
(Epoch 2 / 5) train acc: 0.426000; val_acc: 0.328000
(Iteration 81 / 200) loss: 1.736601
(Iteration 91 / 200) loss: 1.533224
(Iteration 101 / 200) loss: 1.636594
(Iteration 111 / 200) loss: 1.509673
(Epoch 3 / 5) train acc: 0.429000; val_acc: 0.361000
(Iteration 121 / 200) loss: 1.453236
(Iteration 131 / 200) loss: 1.607554
(Iteration 141 / 200) loss: 1.467782
(Iteration 151 / 200) loss: 1.459350
(Epoch 4 / 5) train acc: 0.448000; val_acc: 0.328000
(Iteration 161 / 200) loss: 1.542695
(Iteration 171 / 200) loss: 1.504692
(Iteration 181 / 200) loss: 1.473959
(Iteration 191 / 200) loss: 1.485098
(Epoch 5 / 5) train acc: 0.567000; val_acc: 0.344000
```



13 Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` notebook before completing this part, since those techniques can help you train powerful models.

[ ]:
```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################

################################################################################

# Ensure that dataset keys are correct
assert 'X_train' in data and 'y_train' in data, "Training data not found in␣
 ↪dataset!"
assert 'X_val' in data and 'y_val' in data, "Validation data not found in␣
 ↪dataset!"

# Split the dataset into training and validation sets
num_train = 49000
num_val = 1000

small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'][:num_val],
    'y_val': data['y_val'][:num_val],
}

# Debugging checks
print(f"X_train shape: {small_data['X_train'].shape}")
print(f"y_train shape: {small_data['y_train'].shape}")
print(f"X_val shape: {small_data['X_val'].shape}")
print(f"y_val shape: {small_data['y_val'].shape}")

# Optimized hyperparameter configurations
learning_rates = [1e-3, 5e-4]
weight_scales = [2e-2, 5e-2]
regs = [0.1, 0.2]  # Regularization strengths
hidden_dims = [512, 256, 128]  # Larger hidden layers for capacity
normalization = 'batchnorm'  # Use batch normalization
best_val_acc = 0.0  # Track the best validation accuracy
best_config = None  # Track the best hyperparameter configuration

# Iterate over hyperparameter configurations
for lr in learning_rates:
    for ws in weight_scales:
        for reg in regs:
            print(f"Training model with lr={lr}, ws={ws}, reg={reg}")
```

```
            # Initialize the model without dropout
            model = FullyConnectedNet(hidden_dims, input_dim=3*32*32,␣
↪num_classes=10,
                                      weight_scale=ws, reg=reg,␣
↪normalization=normalization)

            # Initialize the solver with gradient clipping
            solver = Solver(
                model,
                small_data,
                num_epochs=20,  # Faster search with fewer epochs
                batch_size=128,  # Slightly larger batches for stability
                update_rule='adam',
                optim_config={'learning_rate': lr, 'clip_norm': 5.0},  #␣
↪Gradient clipping
                lr_decay=0.95,
                print_every=100
            )

            try:
                # Train the model
                solver.train()

                # Check validation accuracy
                val_acc = solver.check_accuracy(small_data['X_val'],␣
↪small_data['y_val'])
                print(f"Validation accuracy: {val_acc}")

                # Update the best model if validation accuracy improves
                if val_acc > best_val_acc:
                    best_val_acc = val_acc
                    best_model = model
                    best_config = {'learning_rate': lr, 'weight_scale': ws,␣
↪'reg': reg}
            except Exception as e:
                print(f"Error during training with lr={lr}, ws={ws}, reg={reg}:␣
↪{e}")

# Output the best configuration and accuracy
print(f"Best validation accuracy: {best_val_acc}")
print("Best hyperparameter configuration:", best_config)

# Fine-tune the best model with more epochs if needed
if best_model:
    print("Fine-tuning the best model...")
    fine_tune_solver = Solver(
```

```
        best_model,
        small_data,
        num_epochs=50,  # More epochs for fine-tuning
        batch_size=128,
        update_rule='adam',
        optim_config={'learning_rate': best_config['learning_rate'],␣
 ↪'clip_norm': 5.0},
        lr_decay=0.90,
        print_every=100
    )
    fine_tune_solver.train()

    # Final evaluation
    final_train_acc = fine_tune_solver.check_accuracy(small_data['X_train'],␣
 ↪small_data['y_train'])
    final_val_acc = fine_tune_solver.check_accuracy(small_data['X_val'],␣
 ↪small_data['y_val'])
    print(f"Final Training accuracy: {final_train_acc}")
    print(f"Final Validation accuracy: {final_val_acc}")
else:
    print("No valid model was trained.")


################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

```
X_train shape: (49000, 3, 32, 32)
y_train shape: (49000,)
X_val shape: (1000, 3, 32, 32)
y_val shape: (1000,)
Training model with lr=0.001, ws=0.02, reg=0.1
(Iteration 1 / 7640) loss: 37.053115
(Epoch 0 / 20) train acc: 0.190000; val_acc: 0.187000
(Iteration 101 / 7640) loss: 2.350461
(Iteration 201 / 7640) loss: 2.277215
(Iteration 301 / 7640) loss: 2.262963
(Epoch 1 / 20) train acc: 0.372000; val_acc: 0.344000
(Iteration 401 / 7640) loss: 2.237881
(Iteration 501 / 7640) loss: 2.238473
(Iteration 601 / 7640) loss: 2.235126
(Iteration 701 / 7640) loss: 2.323051
(Epoch 2 / 20) train acc: 0.313000; val_acc: 0.338000
(Iteration 801 / 7640) loss: 2.375772
(Iteration 901 / 7640) loss: 2.142891
(Iteration 1001 / 7640) loss: 2.116105
(Iteration 1101 / 7640) loss: 2.195984
```

```
(Epoch 3 / 20) train acc: 0.355000; val_acc: 0.331000
(Iteration 1201 / 7640) loss: 2.243981
(Iteration 1301 / 7640) loss: 2.100598
(Iteration 1401 / 7640) loss: 2.094999
(Iteration 1501 / 7640) loss: 2.110743
(Epoch 4 / 20) train acc: 0.367000; val_acc: 0.357000
(Iteration 1601 / 7640) loss: 2.184827
(Iteration 1701 / 7640) loss: 2.276275
(Iteration 1801 / 7640) loss: 2.104518
(Iteration 1901 / 7640) loss: 2.179351
(Epoch 5 / 20) train acc: 0.345000; val_acc: 0.367000
(Iteration 2001 / 7640) loss: 2.126540
(Iteration 2101 / 7640) loss: 2.088191
(Iteration 2201 / 7640) loss: 2.012448
(Epoch 6 / 20) train acc: 0.400000; val_acc: 0.414000
(Iteration 2301 / 7640) loss: 2.241289
(Iteration 2401 / 7640) loss: 1.968066
(Iteration 2501 / 7640) loss: 2.007288
(Iteration 2601 / 7640) loss: 1.968036
(Epoch 7 / 20) train acc: 0.389000; val_acc: 0.390000
(Iteration 2701 / 7640) loss: 1.873477
(Iteration 2801 / 7640) loss: 2.075384
(Iteration 2901 / 7640) loss: 2.036323
(Iteration 3001 / 7640) loss: 1.780394
(Epoch 8 / 20) train acc: 0.417000; val_acc: 0.417000
(Iteration 3101 / 7640) loss: 1.970517
(Iteration 3201 / 7640) loss: 2.008455
(Iteration 3301 / 7640) loss: 1.946644
(Iteration 3401 / 7640) loss: 1.987604
(Epoch 9 / 20) train acc: 0.427000; val_acc: 0.445000
(Iteration 3501 / 7640) loss: 1.904537
(Iteration 3601 / 7640) loss: 1.949969
(Iteration 3701 / 7640) loss: 1.913553
(Iteration 3801 / 7640) loss: 1.827347
(Epoch 10 / 20) train acc: 0.415000; val_acc: 0.403000
(Iteration 3901 / 7640) loss: 2.074780
(Iteration 4001 / 7640) loss: 1.919738
(Iteration 4101 / 7640) loss: 1.910962
(Iteration 4201 / 7640) loss: 1.806343
(Epoch 11 / 20) train acc: 0.428000; val_acc: 0.432000
(Iteration 4301 / 7640) loss: 2.029126
(Iteration 4401 / 7640) loss: 1.918908
(Iteration 4501 / 7640) loss: 1.912427
(Epoch 12 / 20) train acc: 0.444000; val_acc: 0.444000
(Iteration 4601 / 7640) loss: 1.770973
(Iteration 4701 / 7640) loss: 1.750251
(Iteration 4801 / 7640) loss: 1.910134
(Iteration 4901 / 7640) loss: 1.887651
```

```
(Epoch 13 / 20) train acc: 0.452000; val_acc: 0.469000
(Iteration 5001 / 7640) loss: 1.815057
(Iteration 5101 / 7640) loss: 1.731633
(Iteration 5201 / 7640) loss: 1.589358
(Iteration 5301 / 7640) loss: 1.719876
(Epoch 14 / 20) train acc: 0.430000; val_acc: 0.432000
(Iteration 5401 / 7640) loss: 1.903660
(Iteration 5501 / 7640) loss: 1.656851
(Iteration 5601 / 7640) loss: 1.684541
(Iteration 5701 / 7640) loss: 1.771873
(Epoch 15 / 20) train acc: 0.429000; val_acc: 0.448000
(Iteration 5801 / 7640) loss: 1.777280
(Iteration 5901 / 7640) loss: 1.709418
(Iteration 6001 / 7640) loss: 1.867162
(Iteration 6101 / 7640) loss: 1.664231
(Epoch 16 / 20) train acc: 0.460000; val_acc: 0.459000
(Iteration 6201 / 7640) loss: 1.722643
(Iteration 6301 / 7640) loss: 1.855965
(Iteration 6401 / 7640) loss: 1.678223
(Epoch 17 / 20) train acc: 0.476000; val_acc: 0.464000
(Iteration 6501 / 7640) loss: 1.569977
(Iteration 6601 / 7640) loss: 1.858024
(Iteration 6701 / 7640) loss: 1.696151
(Iteration 6801 / 7640) loss: 1.483778
(Epoch 18 / 20) train acc: 0.460000; val_acc: 0.465000
(Iteration 6901 / 7640) loss: 1.743776
(Iteration 7001 / 7640) loss: 1.715062
(Iteration 7101 / 7640) loss: 1.806389
(Iteration 7201 / 7640) loss: 1.666255
(Epoch 19 / 20) train acc: 0.466000; val_acc: 0.472000
(Iteration 7301 / 7640) loss: 1.670713
(Iteration 7401 / 7640) loss: 1.657659
(Iteration 7501 / 7640) loss: 1.624507
(Iteration 7601 / 7640) loss: 1.569208
(Epoch 20 / 20) train acc: 0.487000; val_acc: 0.461000
Validation accuracy: 0.471
Training model with lr=0.001, ws=0.02, reg=0.2
(Iteration 1 / 7640) loss: 71.775768
(Epoch 0 / 20) train acc: 0.131000; val_acc: 0.145000
(Iteration 101 / 7640) loss: 2.421492
(Iteration 201 / 7640) loss: 2.335700
(Iteration 301 / 7640) loss: 2.377954
(Epoch 1 / 20) train acc: 0.307000; val_acc: 0.287000
(Iteration 401 / 7640) loss: 2.419418
(Iteration 501 / 7640) loss: 2.239276
(Iteration 601 / 7640) loss: 2.412944
(Iteration 701 / 7640) loss: 2.335053
(Epoch 2 / 20) train acc: 0.297000; val_acc: 0.326000
```

```
(Iteration 801 / 7640) loss: 2.326573
(Iteration 901 / 7640) loss: 2.381401
(Iteration 1001 / 7640) loss: 2.331622
(Iteration 1101 / 7640) loss: 2.370836
(Epoch 3 / 20) train acc: 0.281000; val_acc: 0.307000
(Iteration 1201 / 7640) loss: 2.254320
(Iteration 1301 / 7640) loss: 2.336998
(Iteration 1401 / 7640) loss: 2.346028
(Iteration 1501 / 7640) loss: 2.132649
(Epoch 4 / 20) train acc: 0.308000; val_acc: 0.295000
(Iteration 1601 / 7640) loss: 2.272742
(Iteration 1701 / 7640) loss: 2.220941
(Iteration 1801 / 7640) loss: 2.227157
(Iteration 1901 / 7640) loss: 2.266706
(Epoch 5 / 20) train acc: 0.357000; val_acc: 0.348000
(Iteration 2001 / 7640) loss: 2.404674
(Iteration 2101 / 7640) loss: 2.209813
(Iteration 2201 / 7640) loss: 2.268161
(Epoch 6 / 20) train acc: 0.357000; val_acc: 0.352000
(Iteration 2301 / 7640) loss: 2.169630
(Iteration 2401 / 7640) loss: 2.119118
(Iteration 2501 / 7640) loss: 2.123022
(Iteration 2601 / 7640) loss: 2.208379
(Epoch 7 / 20) train acc: 0.358000; val_acc: 0.356000
(Iteration 2701 / 7640) loss: 2.122849
(Iteration 2801 / 7640) loss: 2.116564
(Iteration 2901 / 7640) loss: 2.046051
(Iteration 3001 / 7640) loss: 2.166473
(Epoch 8 / 20) train acc: 0.339000; val_acc: 0.356000
(Iteration 3101 / 7640) loss: 2.144358
(Iteration 3201 / 7640) loss: 2.056104
(Iteration 3301 / 7640) loss: 2.226855
(Iteration 3401 / 7640) loss: 2.018123
(Epoch 9 / 20) train acc: 0.390000; val_acc: 0.373000
(Iteration 3501 / 7640) loss: 2.021867
(Iteration 3601 / 7640) loss: 2.098928
(Iteration 3701 / 7640) loss: 2.168675
(Iteration 3801 / 7640) loss: 2.015695
(Epoch 10 / 20) train acc: 0.377000; val_acc: 0.396000
(Iteration 3901 / 7640) loss: 2.119080
(Iteration 4001 / 7640) loss: 1.994284
(Iteration 4101 / 7640) loss: 1.956645
(Iteration 4201 / 7640) loss: 2.025252
(Epoch 11 / 20) train acc: 0.386000; val_acc: 0.407000
(Iteration 4301 / 7640) loss: 2.004467
(Iteration 4401 / 7640) loss: 2.048892
(Iteration 4501 / 7640) loss: 2.031117
(Epoch 12 / 20) train acc: 0.411000; val_acc: 0.400000
```

```
(Iteration 4601 / 7640) loss: 1.932754
(Iteration 4701 / 7640) loss: 1.927626
(Iteration 4801 / 7640) loss: 1.944643
(Iteration 4901 / 7640) loss: 1.945926
(Epoch 13 / 20) train acc: 0.428000; val_acc: 0.422000
(Iteration 5001 / 7640) loss: 2.050431
(Iteration 5101 / 7640) loss: 1.921238
(Iteration 5201 / 7640) loss: 1.894373
(Iteration 5301 / 7640) loss: 1.904199
(Epoch 14 / 20) train acc: 0.411000; val_acc: 0.440000
(Iteration 5401 / 7640) loss: 1.773468
(Iteration 5501 / 7640) loss: 1.960075
(Iteration 5601 / 7640) loss: 2.059485
(Iteration 5701 / 7640) loss: 1.994383
(Epoch 15 / 20) train acc: 0.396000; val_acc: 0.420000
(Iteration 5801 / 7640) loss: 1.994204
(Iteration 5901 / 7640) loss: 2.043924
(Iteration 6001 / 7640) loss: 1.973428
(Iteration 6101 / 7640) loss: 1.879480
(Epoch 16 / 20) train acc: 0.422000; val_acc: 0.399000
(Iteration 6201 / 7640) loss: 1.822617
(Iteration 6301 / 7640) loss: 2.066060
(Iteration 6401 / 7640) loss: 1.754715
(Epoch 17 / 20) train acc: 0.446000; val_acc: 0.452000
(Iteration 6501 / 7640) loss: 1.857154
(Iteration 6601 / 7640) loss: 1.814260
(Iteration 6701 / 7640) loss: 1.926756
(Iteration 6801 / 7640) loss: 1.874674
(Epoch 18 / 20) train acc: 0.430000; val_acc: 0.454000
(Iteration 6901 / 7640) loss: 1.807579
(Iteration 7001 / 7640) loss: 1.865170
(Iteration 7101 / 7640) loss: 1.756564
(Iteration 7201 / 7640) loss: 1.826764
(Epoch 19 / 20) train acc: 0.441000; val_acc: 0.438000
(Iteration 7301 / 7640) loss: 1.582384
(Iteration 7401 / 7640) loss: 1.925219
(Iteration 7501 / 7640) loss: 1.651097
(Iteration 7601 / 7640) loss: 1.739206
(Epoch 20 / 20) train acc: 0.446000; val_acc: 0.433000
Validation accuracy: 0.427
Training model with lr=0.001, ws=0.05, reg=0.1
(Iteration 1 / 7640) loss: 219.736883
(Epoch 0 / 20) train acc: 0.137000; val_acc: 0.142000
(Iteration 101 / 7640) loss: 10.514985
(Iteration 201 / 7640) loss: 2.643959
(Iteration 301 / 7640) loss: 2.274445
(Epoch 1 / 20) train acc: 0.315000; val_acc: 0.336000
(Iteration 401 / 7640) loss: 2.250285
```

```
(Iteration 501 / 7640) loss: 2.162316
(Iteration 601 / 7640) loss: 2.199693
(Iteration 701 / 7640) loss: 2.111041
(Epoch 2 / 20) train acc: 0.351000; val_acc: 0.371000
(Iteration 801 / 7640) loss: 2.095162
(Iteration 901 / 7640) loss: 2.209530
(Iteration 1001 / 7640) loss: 2.162115
(Iteration 1101 / 7640) loss: 2.241372
(Epoch 3 / 20) train acc: 0.332000; val_acc: 0.339000
(Iteration 1201 / 7640) loss: 2.131085
(Iteration 1301 / 7640) loss: 2.052899
(Iteration 1401 / 7640) loss: 2.160649
(Iteration 1501 / 7640) loss: 2.183511
(Epoch 4 / 20) train acc: 0.303000; val_acc: 0.309000
(Iteration 1601 / 7640) loss: 2.173910
(Iteration 1701 / 7640) loss: 2.097313
(Iteration 1801 / 7640) loss: 2.173603
(Iteration 1901 / 7640) loss: 2.013094
(Epoch 5 / 20) train acc: 0.339000; val_acc: 0.378000
(Iteration 2001 / 7640) loss: 2.178747
(Iteration 2101 / 7640) loss: 2.045629
(Iteration 2201 / 7640) loss: 2.088480
(Epoch 6 / 20) train acc: 0.350000; val_acc: 0.381000
(Iteration 2301 / 7640) loss: 2.022018
(Iteration 2401 / 7640) loss: 1.948752
(Iteration 2501 / 7640) loss: 1.964516
(Iteration 2601 / 7640) loss: 2.014671
(Epoch 7 / 20) train acc: 0.341000; val_acc: 0.406000
(Iteration 2701 / 7640) loss: 1.970055
(Iteration 2801 / 7640) loss: 2.064232
(Iteration 2901 / 7640) loss: 2.232343
(Iteration 3001 / 7640) loss: 1.919220
(Epoch 8 / 20) train acc: 0.393000; val_acc: 0.411000
(Iteration 3101 / 7640) loss: 1.932516
(Iteration 3201 / 7640) loss: 2.043253
(Iteration 3301 / 7640) loss: 2.000173
(Iteration 3401 / 7640) loss: 1.827605
(Epoch 9 / 20) train acc: 0.397000; val_acc: 0.421000
(Iteration 3501 / 7640) loss: 1.941273
(Iteration 3601 / 7640) loss: 1.971099
(Iteration 3701 / 7640) loss: 1.941869
(Iteration 3801 / 7640) loss: 1.932531
(Epoch 10 / 20) train acc: 0.431000; val_acc: 0.425000
(Iteration 3901 / 7640) loss: 2.037336
(Iteration 4001 / 7640) loss: 1.944728
(Iteration 4101 / 7640) loss: 1.740606
(Iteration 4201 / 7640) loss: 2.023647
(Epoch 11 / 20) train acc: 0.421000; val_acc: 0.389000
```

```
(Iteration 4301 / 7640) loss: 1.783193
(Iteration 4401 / 7640) loss: 1.865429
(Iteration 4501 / 7640) loss: 1.824347
(Epoch 12 / 20) train acc: 0.436000; val_acc: 0.427000
(Iteration 4601 / 7640) loss: 2.011483
(Iteration 4701 / 7640) loss: 1.964606
(Iteration 4801 / 7640) loss: 1.725580
(Iteration 4901 / 7640) loss: 1.908534
(Epoch 13 / 20) train acc: 0.436000; val_acc: 0.414000
(Iteration 5001 / 7640) loss: 1.794465
(Iteration 5101 / 7640) loss: 1.762873
(Iteration 5201 / 7640) loss: 1.675427
(Iteration 5301 / 7640) loss: 1.815651
(Epoch 14 / 20) train acc: 0.460000; val_acc: 0.459000
(Iteration 5401 / 7640) loss: 1.838094
(Iteration 5501 / 7640) loss: 1.628667
(Iteration 5601 / 7640) loss: 1.758129
(Iteration 5701 / 7640) loss: 1.685370
(Epoch 15 / 20) train acc: 0.451000; val_acc: 0.434000
(Iteration 5801 / 7640) loss: 1.818978
(Iteration 5901 / 7640) loss: 1.704568
(Iteration 6001 / 7640) loss: 1.787493
(Iteration 6101 / 7640) loss: 1.594946
(Epoch 16 / 20) train acc: 0.468000; val_acc: 0.456000
(Iteration 6201 / 7640) loss: 1.697998
(Iteration 6301 / 7640) loss: 1.621925
(Iteration 6401 / 7640) loss: 1.692727
(Epoch 17 / 20) train acc: 0.475000; val_acc: 0.461000
(Iteration 6501 / 7640) loss: 1.708178
(Iteration 6601 / 7640) loss: 1.603912
(Iteration 6701 / 7640) loss: 1.701760
(Iteration 6801 / 7640) loss: 1.754263
(Epoch 18 / 20) train acc: 0.453000; val_acc: 0.465000
(Iteration 6901 / 7640) loss: 1.504440
(Iteration 7001 / 7640) loss: 1.650844
(Iteration 7101 / 7640) loss: 1.689480
(Iteration 7201 / 7640) loss: 1.714925
(Epoch 19 / 20) train acc: 0.464000; val_acc: 0.447000
(Iteration 7301 / 7640) loss: 1.650856
(Iteration 7401 / 7640) loss: 1.733157
(Iteration 7501 / 7640) loss: 1.717699
(Iteration 7601 / 7640) loss: 1.827036
(Epoch 20 / 20) train acc: 0.501000; val_acc: 0.466000
Validation accuracy: 0.466
Training model with lr=0.001, ws=0.05, reg=0.2
(Iteration 1 / 7640) loss: 436.690169
(Epoch 0 / 20) train acc: 0.098000; val_acc: 0.111000
(Iteration 101 / 7640) loss: 18.298689
```

```
(Iteration 201 / 7640) loss: 3.013095
(Iteration 301 / 7640) loss: 2.387186
(Epoch 1 / 20) train acc: 0.314000; val_acc: 0.318000
(Iteration 401 / 7640) loss: 2.290406
(Iteration 501 / 7640) loss: 2.262268
(Iteration 601 / 7640) loss: 2.279530
(Iteration 701 / 7640) loss: 2.362880
(Epoch 2 / 20) train acc: 0.329000; val_acc: 0.331000
(Iteration 801 / 7640) loss: 2.246888
(Iteration 901 / 7640) loss: 2.265478
(Iteration 1001 / 7640) loss: 2.262045
(Iteration 1101 / 7640) loss: 2.268522
(Epoch 3 / 20) train acc: 0.299000; val_acc: 0.312000
(Iteration 1201 / 7640) loss: 2.299738
(Iteration 1301 / 7640) loss: 2.244076
(Iteration 1401 / 7640) loss: 2.318872
(Iteration 1501 / 7640) loss: 2.412490
(Epoch 4 / 20) train acc: 0.332000; val_acc: 0.320000
(Iteration 1601 / 7640) loss: 2.272553
(Iteration 1701 / 7640) loss: 2.206759
(Iteration 1801 / 7640) loss: 2.213082
(Iteration 1901 / 7640) loss: 2.288573
(Epoch 5 / 20) train acc: 0.311000; val_acc: 0.318000
(Iteration 2001 / 7640) loss: 2.303975
(Iteration 2101 / 7640) loss: 2.293291
(Iteration 2201 / 7640) loss: 2.238695
(Epoch 6 / 20) train acc: 0.339000; val_acc: 0.336000
(Iteration 2301 / 7640) loss: 2.308621
(Iteration 2401 / 7640) loss: 2.183256
(Iteration 2501 / 7640) loss: 2.180750
(Iteration 2601 / 7640) loss: 2.280819
(Epoch 7 / 20) train acc: 0.360000; val_acc: 0.360000
(Iteration 2701 / 7640) loss: 2.166041
(Iteration 2801 / 7640) loss: 2.262252
(Iteration 2901 / 7640) loss: 2.300337
(Iteration 3001 / 7640) loss: 2.186093
(Epoch 8 / 20) train acc: 0.363000; val_acc: 0.364000
(Iteration 3101 / 7640) loss: 2.237843
(Iteration 3201 / 7640) loss: 2.081913
(Iteration 3301 / 7640) loss: 2.194698
(Iteration 3401 / 7640) loss: 2.141210
(Epoch 9 / 20) train acc: 0.350000; val_acc: 0.365000
(Iteration 3501 / 7640) loss: 2.098669
(Iteration 3601 / 7640) loss: 2.055623
(Iteration 3701 / 7640) loss: 2.090170
(Iteration 3801 / 7640) loss: 2.018240
(Epoch 10 / 20) train acc: 0.362000; val_acc: 0.381000
(Iteration 3901 / 7640) loss: 2.072889
```

```
(Iteration 4001 / 7640) loss: 2.200291
(Iteration 4101 / 7640) loss: 1.973672
(Iteration 4201 / 7640) loss: 2.065696
(Epoch 11 / 20) train acc: 0.391000; val_acc: 0.411000
(Iteration 4301 / 7640) loss: 2.147988
(Iteration 4401 / 7640) loss: 1.944207
(Iteration 4501 / 7640) loss: 1.960086
(Epoch 12 / 20) train acc: 0.385000; val_acc: 0.372000
(Iteration 4601 / 7640) loss: 2.013227
(Iteration 4701 / 7640) loss: 1.993427
(Iteration 4801 / 7640) loss: 2.011831
(Iteration 4901 / 7640) loss: 1.931530
(Epoch 13 / 20) train acc: 0.376000; val_acc: 0.409000
(Iteration 5001 / 7640) loss: 2.119909
(Iteration 5101 / 7640) loss: 2.035286
(Iteration 5201 / 7640) loss: 1.918886
(Iteration 5301 / 7640) loss: 1.939748
(Epoch 14 / 20) train acc: 0.407000; val_acc: 0.414000
(Iteration 5401 / 7640) loss: 1.897239
(Iteration 5501 / 7640) loss: 1.976688
(Iteration 5601 / 7640) loss: 1.758358
(Iteration 5701 / 7640) loss: 2.021105
(Epoch 15 / 20) train acc: 0.432000; val_acc: 0.412000
(Iteration 5801 / 7640) loss: 1.940628
(Iteration 5901 / 7640) loss: 1.809829
(Iteration 6001 / 7640) loss: 1.940355
(Iteration 6101 / 7640) loss: 2.081408
(Epoch 16 / 20) train acc: 0.411000; val_acc: 0.397000
(Iteration 6201 / 7640) loss: 1.888362
(Iteration 6301 / 7640) loss: 1.974092
(Iteration 6401 / 7640) loss: 1.963730
(Epoch 17 / 20) train acc: 0.403000; val_acc: 0.418000
(Iteration 6501 / 7640) loss: 1.919847
(Iteration 6601 / 7640) loss: 1.840065
(Iteration 6701 / 7640) loss: 1.864555
(Iteration 6801 / 7640) loss: 1.846397
(Epoch 18 / 20) train acc: 0.431000; val_acc: 0.434000
(Iteration 6901 / 7640) loss: 1.860437
(Iteration 7001 / 7640) loss: 1.997929
(Iteration 7101 / 7640) loss: 1.890910
(Iteration 7201 / 7640) loss: 1.981258
(Epoch 19 / 20) train acc: 0.474000; val_acc: 0.453000
(Iteration 7301 / 7640) loss: 1.706091
(Iteration 7401 / 7640) loss: 1.940162
(Iteration 7501 / 7640) loss: 1.817805
(Iteration 7601 / 7640) loss: 1.965315
(Epoch 20 / 20) train acc: 0.448000; val_acc: 0.459000
Validation accuracy: 0.459
```

```
Training model with lr=0.0005, ws=0.02, reg=0.1
(Iteration 1 / 7640) loss: 37.074226
(Epoch 0 / 20) train acc: 0.162000; val_acc: 0.159000
(Iteration 101 / 7640) loss: 2.648917
(Iteration 201 / 7640) loss: 2.073790
(Iteration 301 / 7640) loss: 2.080681
(Epoch 1 / 20) train acc: 0.407000; val_acc: 0.388000
(Iteration 401 / 7640) loss: 2.142654
(Iteration 501 / 7640) loss: 2.056224
(Iteration 601 / 7640) loss: 1.967427
(Iteration 701 / 7640) loss: 1.934720
(Epoch 2 / 20) train acc: 0.366000; val_acc: 0.394000
(Iteration 801 / 7640) loss: 2.004302
(Iteration 901 / 7640) loss: 1.967542
(Iteration 1001 / 7640) loss: 2.090345
(Iteration 1101 / 7640) loss: 1.935147
(Epoch 3 / 20) train acc: 0.403000; val_acc: 0.399000
(Iteration 1201 / 7640) loss: 2.051290
(Iteration 1301 / 7640) loss: 1.916075
(Iteration 1401 / 7640) loss: 2.100677
(Iteration 1501 / 7640) loss: 1.986532
(Epoch 4 / 20) train acc: 0.428000; val_acc: 0.413000
(Iteration 1601 / 7640) loss: 1.958041
(Iteration 1701 / 7640) loss: 1.850240
(Iteration 1801 / 7640) loss: 1.958426
(Iteration 1901 / 7640) loss: 2.110655
(Epoch 5 / 20) train acc: 0.418000; val_acc: 0.403000
(Iteration 2001 / 7640) loss: 1.867353
(Iteration 2101 / 7640) loss: 1.745152
(Iteration 2201 / 7640) loss: 1.903735
(Epoch 6 / 20) train acc: 0.437000; val_acc: 0.418000
(Iteration 2301 / 7640) loss: 1.924077
(Iteration 2401 / 7640) loss: 1.844173
(Iteration 2501 / 7640) loss: 1.939731
(Iteration 2601 / 7640) loss: 1.902681
(Epoch 7 / 20) train acc: 0.423000; val_acc: 0.439000
(Iteration 2701 / 7640) loss: 1.653353
(Iteration 2801 / 7640) loss: 1.848841
(Iteration 2901 / 7640) loss: 1.837019
(Iteration 3001 / 7640) loss: 1.578193
(Epoch 8 / 20) train acc: 0.446000; val_acc: 0.445000
(Iteration 3101 / 7640) loss: 1.951288
(Iteration 3201 / 7640) loss: 1.888997
(Iteration 3301 / 7640) loss: 1.756006
(Iteration 3401 / 7640) loss: 1.777829
(Epoch 9 / 20) train acc: 0.448000; val_acc: 0.423000
(Iteration 3501 / 7640) loss: 1.816929
(Iteration 3601 / 7640) loss: 1.696113
```

```
(Iteration 3701 / 7640) loss: 1.763948
(Iteration 3801 / 7640) loss: 1.854665
(Epoch 10 / 20) train acc: 0.459000; val_acc: 0.433000
(Iteration 3901 / 7640) loss: 1.893626
(Iteration 4001 / 7640) loss: 1.775010
(Iteration 4101 / 7640) loss: 1.618995
(Iteration 4201 / 7640) loss: 1.734212
(Epoch 11 / 20) train acc: 0.515000; val_acc: 0.473000
(Iteration 4301 / 7640) loss: 1.673626
(Iteration 4401 / 7640) loss: 1.650851
(Iteration 4501 / 7640) loss: 1.859515
(Epoch 12 / 20) train acc: 0.463000; val_acc: 0.448000
(Iteration 4601 / 7640) loss: 1.734636
(Iteration 4701 / 7640) loss: 1.659620
(Iteration 4801 / 7640) loss: 1.699257
(Iteration 4901 / 7640) loss: 1.705550
(Epoch 13 / 20) train acc: 0.480000; val_acc: 0.467000
(Iteration 5001 / 7640) loss: 1.813373
(Iteration 5101 / 7640) loss: 1.730530
(Iteration 5201 / 7640) loss: 1.776567
(Iteration 5301 / 7640) loss: 1.470639
(Epoch 14 / 20) train acc: 0.486000; val_acc: 0.486000
(Iteration 5401 / 7640) loss: 1.670341
(Iteration 5501 / 7640) loss: 1.566366
(Iteration 5601 / 7640) loss: 1.550935
(Iteration 5701 / 7640) loss: 1.728077
(Epoch 15 / 20) train acc: 0.501000; val_acc: 0.486000
(Iteration 5801 / 7640) loss: 1.590102
(Iteration 5901 / 7640) loss: 1.535443
(Iteration 6001 / 7640) loss: 1.499257
(Iteration 6101 / 7640) loss: 1.657314
(Epoch 16 / 20) train acc: 0.534000; val_acc: 0.514000
(Iteration 6201 / 7640) loss: 1.588927
(Iteration 6301 / 7640) loss: 1.774876
(Iteration 6401 / 7640) loss: 1.609158
(Epoch 17 / 20) train acc: 0.508000; val_acc: 0.484000
(Iteration 6501 / 7640) loss: 1.632782
(Iteration 6601 / 7640) loss: 1.654609
(Iteration 6701 / 7640) loss: 1.547007
(Iteration 6801 / 7640) loss: 1.470937
(Epoch 18 / 20) train acc: 0.527000; val_acc: 0.486000
(Iteration 6901 / 7640) loss: 1.495216
(Iteration 7001 / 7640) loss: 1.562378
(Iteration 7101 / 7640) loss: 1.556255
(Iteration 7201 / 7640) loss: 1.598786
(Epoch 19 / 20) train acc: 0.527000; val_acc: 0.514000
(Iteration 7301 / 7640) loss: 1.446461
(Iteration 7401 / 7640) loss: 1.639213
```

```
(Iteration 7501 / 7640) loss: 1.455836
(Iteration 7601 / 7640) loss: 1.514924
(Epoch 20 / 20) train acc: 0.552000; val_acc: 0.509000
Validation accuracy: 0.472
Training model with lr=0.0005, ws=0.02, reg=0.2
(Iteration 1 / 7640) loss: 71.751920
(Epoch 0 / 20) train acc: 0.116000; val_acc: 0.120000
(Iteration 101 / 7640) loss: 3.216626
(Iteration 201 / 7640) loss: 2.145224
(Iteration 301 / 7640) loss: 2.204134
(Epoch 1 / 20) train acc: 0.333000; val_acc: 0.377000
(Iteration 401 / 7640) loss: 2.066607
(Iteration 501 / 7640) loss: 2.069206
(Iteration 601 / 7640) loss: 2.117460
(Iteration 701 / 7640) loss: 2.107070
(Epoch 2 / 20) train acc: 0.350000; val_acc: 0.336000
(Iteration 801 / 7640) loss: 2.099022
(Iteration 901 / 7640) loss: 2.192851
(Iteration 1001 / 7640) loss: 2.240036
(Iteration 1101 / 7640) loss: 2.132569
(Epoch 3 / 20) train acc: 0.352000; val_acc: 0.373000
(Iteration 1201 / 7640) loss: 2.117924
(Iteration 1301 / 7640) loss: 2.092722
(Iteration 1401 / 7640) loss: 2.128703
(Iteration 1501 / 7640) loss: 2.039933
(Epoch 4 / 20) train acc: 0.373000; val_acc: 0.359000
(Iteration 1601 / 7640) loss: 2.049060
(Iteration 1701 / 7640) loss: 2.069466
(Iteration 1801 / 7640) loss: 1.960854
(Iteration 1901 / 7640) loss: 2.257464
(Epoch 5 / 20) train acc: 0.350000; val_acc: 0.355000
(Iteration 2001 / 7640) loss: 2.045312
(Iteration 2101 / 7640) loss: 2.186255
(Iteration 2201 / 7640) loss: 2.022469
(Epoch 6 / 20) train acc: 0.351000; val_acc: 0.388000
(Iteration 2301 / 7640) loss: 2.074474
(Iteration 2401 / 7640) loss: 2.048031
(Iteration 2501 / 7640) loss: 2.017069
(Iteration 2601 / 7640) loss: 2.069724
(Epoch 7 / 20) train acc: 0.404000; val_acc: 0.403000
(Iteration 2701 / 7640) loss: 1.982738
(Iteration 2801 / 7640) loss: 2.035182
(Iteration 2901 / 7640) loss: 2.058254
(Iteration 3001 / 7640) loss: 2.013535
(Epoch 8 / 20) train acc: 0.390000; val_acc: 0.422000
(Iteration 3101 / 7640) loss: 2.064325
(Iteration 3201 / 7640) loss: 2.071780
(Iteration 3301 / 7640) loss: 1.982267
```

```
(Iteration 3401 / 7640) loss: 1.747820
(Epoch 9 / 20) train acc: 0.401000; val_acc: 0.399000
(Iteration 3501 / 7640) loss: 1.928847
(Iteration 3601 / 7640) loss: 2.032235
(Iteration 3701 / 7640) loss: 1.931655
(Iteration 3801 / 7640) loss: 1.887988
(Epoch 10 / 20) train acc: 0.412000; val_acc: 0.420000
(Iteration 3901 / 7640) loss: 1.794162
(Iteration 4001 / 7640) loss: 1.972174
(Iteration 4101 / 7640) loss: 1.879138
(Iteration 4201 / 7640) loss: 1.856269
(Epoch 11 / 20) train acc: 0.422000; val_acc: 0.460000
(Iteration 4301 / 7640) loss: 1.927887
(Iteration 4401 / 7640) loss: 1.795819
(Iteration 4501 / 7640) loss: 1.848070
(Epoch 12 / 20) train acc: 0.445000; val_acc: 0.420000
(Iteration 4601 / 7640) loss: 1.861106
(Iteration 4701 / 7640) loss: 1.782145
(Iteration 4801 / 7640) loss: 1.756189
(Iteration 4901 / 7640) loss: 1.869596
(Epoch 13 / 20) train acc: 0.451000; val_acc: 0.445000
(Iteration 5001 / 7640) loss: 1.790403
(Iteration 5101 / 7640) loss: 1.770248
(Iteration 5201 / 7640) loss: 1.998237
(Iteration 5301 / 7640) loss: 1.961089
(Epoch 14 / 20) train acc: 0.445000; val_acc: 0.448000
(Iteration 5401 / 7640) loss: 1.783802
(Iteration 5501 / 7640) loss: 1.642605
(Iteration 5601 / 7640) loss: 1.813479
(Iteration 5701 / 7640) loss: 1.751388
(Epoch 15 / 20) train acc: 0.479000; val_acc: 0.465000
(Iteration 5801 / 7640) loss: 1.748967
(Iteration 5901 / 7640) loss: 1.609571
(Iteration 6001 / 7640) loss: 1.709868
(Iteration 6101 / 7640) loss: 1.643560
(Epoch 16 / 20) train acc: 0.492000; val_acc: 0.470000
(Iteration 6201 / 7640) loss: 1.679941
(Iteration 6301 / 7640) loss: 1.765365
(Iteration 6401 / 7640) loss: 1.622446
(Epoch 17 / 20) train acc: 0.498000; val_acc: 0.484000
(Iteration 6501 / 7640) loss: 1.668747
(Iteration 6601 / 7640) loss: 1.716179
(Iteration 6701 / 7640) loss: 1.703739
(Iteration 6801 / 7640) loss: 1.794712
(Epoch 18 / 20) train acc: 0.471000; val_acc: 0.471000
(Iteration 6901 / 7640) loss: 1.746185
(Iteration 7001 / 7640) loss: 1.647476
(Iteration 7101 / 7640) loss: 1.743243
```

```
(Iteration 7201 / 7640) loss: 1.838119
(Epoch 19 / 20) train acc: 0.476000; val_acc: 0.472000
(Iteration 7301 / 7640) loss: 1.734519
(Iteration 7401 / 7640) loss: 1.779577
(Iteration 7501 / 7640) loss: 1.582764
(Iteration 7601 / 7640) loss: 1.788279
(Epoch 20 / 20) train acc: 0.469000; val_acc: 0.496000
Validation accuracy: 0.496
Training model with lr=0.0005, ws=0.05, reg=0.1
(Iteration 1 / 7640) loss: 219.765516
(Epoch 0 / 20) train acc: 0.106000; val_acc: 0.096000
(Iteration 101 / 7640) loss: 46.716899
(Iteration 201 / 7640) loss: 11.941328
(Iteration 301 / 7640) loss: 4.318219
(Epoch 1 / 20) train acc: 0.375000; val_acc: 0.350000
(Iteration 401 / 7640) loss: 2.683348
(Iteration 501 / 7640) loss: 2.056465
(Iteration 601 / 7640) loss: 1.948074
(Iteration 701 / 7640) loss: 2.167165
(Epoch 2 / 20) train acc: 0.332000; val_acc: 0.377000
(Iteration 801 / 7640) loss: 1.952445
(Iteration 901 / 7640) loss: 1.989644
(Iteration 1001 / 7640) loss: 2.017790
(Iteration 1101 / 7640) loss: 2.182473
(Epoch 3 / 20) train acc: 0.377000; val_acc: 0.379000
(Iteration 1201 / 7640) loss: 1.880465
(Iteration 1301 / 7640) loss: 1.976955
(Iteration 1401 / 7640) loss: 1.849007
(Iteration 1501 / 7640) loss: 2.173123
(Epoch 4 / 20) train acc: 0.425000; val_acc: 0.398000
(Iteration 1601 / 7640) loss: 1.916120
(Iteration 1701 / 7640) loss: 1.925349
(Iteration 1801 / 7640) loss: 1.865666
(Iteration 1901 / 7640) loss: 1.902794
(Epoch 5 / 20) train acc: 0.443000; val_acc: 0.448000
(Iteration 2001 / 7640) loss: 1.999546
(Iteration 2101 / 7640) loss: 1.946397
(Iteration 2201 / 7640) loss: 1.882833
(Epoch 6 / 20) train acc: 0.434000; val_acc: 0.420000
(Iteration 2301 / 7640) loss: 2.018494
(Iteration 2401 / 7640) loss: 1.941798
(Iteration 2501 / 7640) loss: 2.052199
(Iteration 2601 / 7640) loss: 1.925917
(Epoch 7 / 20) train acc: 0.439000; val_acc: 0.456000
(Iteration 2701 / 7640) loss: 1.995277
(Iteration 2801 / 7640) loss: 1.830329
(Iteration 2901 / 7640) loss: 1.869261
(Iteration 3001 / 7640) loss: 1.858369
```

```
(Epoch 8 / 20) train acc: 0.459000; val_acc: 0.442000
(Iteration 3101 / 7640) loss: 1.885204
(Iteration 3201 / 7640) loss: 1.938201
(Iteration 3301 / 7640) loss: 1.858959
(Iteration 3401 / 7640) loss: 1.708282
(Epoch 9 / 20) train acc: 0.460000; val_acc: 0.458000
(Iteration 3501 / 7640) loss: 1.795133
(Iteration 3601 / 7640) loss: 1.792532
(Iteration 3701 / 7640) loss: 1.695158
(Iteration 3801 / 7640) loss: 1.837034
(Epoch 10 / 20) train acc: 0.489000; val_acc: 0.458000
(Iteration 3901 / 7640) loss: 1.771438
(Iteration 4001 / 7640) loss: 1.655020
(Iteration 4101 / 7640) loss: 1.901537
(Iteration 4201 / 7640) loss: 1.660757
(Epoch 11 / 20) train acc: 0.466000; val_acc: 0.467000
(Iteration 4301 / 7640) loss: 1.646256
(Iteration 4401 / 7640) loss: 1.673271
(Iteration 4501 / 7640) loss: 1.826255
(Epoch 12 / 20) train acc: 0.461000; val_acc: 0.459000
(Iteration 4601 / 7640) loss: 1.752989
(Iteration 4701 / 7640) loss: 1.807253
(Iteration 4801 / 7640) loss: 1.825338
(Iteration 4901 / 7640) loss: 1.831293
(Epoch 13 / 20) train acc: 0.503000; val_acc: 0.469000
(Iteration 5001 / 7640) loss: 1.575401
(Iteration 5101 / 7640) loss: 1.642544
(Iteration 5201 / 7640) loss: 1.709856
(Iteration 5301 / 7640) loss: 1.647516
(Epoch 14 / 20) train acc: 0.505000; val_acc: 0.477000
(Iteration 5401 / 7640) loss: 1.654015
(Iteration 5501 / 7640) loss: 1.747449
(Iteration 5601 / 7640) loss: 1.630582
(Iteration 5701 / 7640) loss: 1.778722
(Epoch 15 / 20) train acc: 0.534000; val_acc: 0.499000
(Iteration 5801 / 7640) loss: 1.744057
(Iteration 5901 / 7640) loss: 1.725563
(Iteration 6001 / 7640) loss: 1.793912
(Iteration 6101 / 7640) loss: 1.614610
(Epoch 16 / 20) train acc: 0.519000; val_acc: 0.495000
(Iteration 6201 / 7640) loss: 1.474489
(Iteration 6301 / 7640) loss: 1.867005
(Iteration 6401 / 7640) loss: 1.575457
(Epoch 17 / 20) train acc: 0.531000; val_acc: 0.509000
(Iteration 6501 / 7640) loss: 1.654947
(Iteration 6601 / 7640) loss: 1.537901
(Iteration 6701 / 7640) loss: 1.700985
(Iteration 6801 / 7640) loss: 1.553424
```

```
(Epoch 18 / 20) train acc: 0.518000; val_acc: 0.495000
(Iteration 6901 / 7640) loss: 1.687228
(Iteration 7001 / 7640) loss: 1.504283
(Iteration 7101 / 7640) loss: 1.729221
(Iteration 7201 / 7640) loss: 1.619351
(Epoch 19 / 20) train acc: 0.567000; val_acc: 0.516000
(Iteration 7301 / 7640) loss: 1.611517
(Iteration 7401 / 7640) loss: 1.617784
(Iteration 7501 / 7640) loss: 1.600136
(Iteration 7601 / 7640) loss: 1.448708
(Epoch 20 / 20) train acc: 0.533000; val_acc: 0.512000
Validation accuracy: 0.504
Training model with lr=0.0005, ws=0.05, reg=0.2
(Iteration 1 / 7640) loss: 436.383829
(Epoch 0 / 20) train acc: 0.109000; val_acc: 0.102000
(Iteration 101 / 7640) loss: 89.124304
(Iteration 201 / 7640) loss: 20.875633
(Iteration 301 / 7640) loss: 6.270121
(Epoch 1 / 20) train acc: 0.358000; val_acc: 0.347000
(Iteration 401 / 7640) loss: 3.140576
(Iteration 501 / 7640) loss: 2.339225
(Iteration 601 / 7640) loss: 2.253530
(Iteration 701 / 7640) loss: 2.321959
(Epoch 2 / 20) train acc: 0.361000; val_acc: 0.361000
(Iteration 801 / 7640) loss: 2.037987
(Iteration 901 / 7640) loss: 2.149919
(Iteration 1001 / 7640) loss: 2.168029
(Iteration 1101 / 7640) loss: 2.097344
(Epoch 3 / 20) train acc: 0.363000; val_acc: 0.353000
(Iteration 1201 / 7640) loss: 2.075224
(Iteration 1301 / 7640) loss: 2.054978
(Iteration 1401 / 7640) loss: 2.169432
(Iteration 1501 / 7640) loss: 2.164924
(Epoch 4 / 20) train acc: 0.354000; val_acc: 0.359000
(Iteration 1601 / 7640) loss: 2.194936
(Iteration 1701 / 7640) loss: 2.092238
(Iteration 1801 / 7640) loss: 1.947927
(Iteration 1901 / 7640) loss: 2.048431
(Epoch 5 / 20) train acc: 0.364000; val_acc: 0.373000
(Iteration 2001 / 7640) loss: 2.084587
(Iteration 2101 / 7640) loss: 2.101547
(Iteration 2201 / 7640) loss: 2.154021
(Epoch 6 / 20) train acc: 0.343000; val_acc: 0.347000
(Iteration 2301 / 7640) loss: 2.218647
(Iteration 2401 / 7640) loss: 2.052370
(Iteration 2501 / 7640) loss: 1.998282
(Iteration 2601 / 7640) loss: 1.973815
(Epoch 7 / 20) train acc: 0.399000; val_acc: 0.405000
```

```
(Iteration 2701 / 7640) loss: 2.195093
(Iteration 2801 / 7640) loss: 1.931587
(Iteration 2901 / 7640) loss: 2.059685
(Iteration 3001 / 7640) loss: 1.912056
(Epoch 8 / 20) train acc: 0.361000; val_acc: 0.410000
(Iteration 3101 / 7640) loss: 1.897965
(Iteration 3201 / 7640) loss: 1.810105
(Iteration 3301 / 7640) loss: 1.854309
(Iteration 3401 / 7640) loss: 2.061910
(Epoch 9 / 20) train acc: 0.372000; val_acc: 0.390000
(Iteration 3501 / 7640) loss: 1.955679
(Iteration 3601 / 7640) loss: 1.787995
(Iteration 3701 / 7640) loss: 1.949800
(Iteration 3801 / 7640) loss: 1.898788
(Epoch 10 / 20) train acc: 0.408000; val_acc: 0.416000
(Iteration 3901 / 7640) loss: 2.002471
(Iteration 4001 / 7640) loss: 1.874139
(Iteration 4101 / 7640) loss: 1.662328
(Iteration 4201 / 7640) loss: 2.093645
(Epoch 11 / 20) train acc: 0.451000; val_acc: 0.432000
(Iteration 4301 / 7640) loss: 1.953200
(Iteration 4401 / 7640) loss: 1.969772
(Iteration 4501 / 7640) loss: 2.079482
(Epoch 12 / 20) train acc: 0.444000; val_acc: 0.448000
(Iteration 4601 / 7640) loss: 1.835996
(Iteration 4701 / 7640) loss: 1.897803
(Iteration 4801 / 7640) loss: 1.873095
(Iteration 4901 / 7640) loss: 1.906980
(Epoch 13 / 20) train acc: 0.424000; val_acc: 0.451000
(Iteration 5001 / 7640) loss: 1.855073
(Iteration 5101 / 7640) loss: 1.891701
(Iteration 5201 / 7640) loss: 1.850525
(Iteration 5301 / 7640) loss: 1.872469
(Epoch 14 / 20) train acc: 0.464000; val_acc: 0.449000
(Iteration 5401 / 7640) loss: 1.791294
(Iteration 5501 / 7640) loss: 1.699614
(Iteration 5601 / 7640) loss: 1.724384
(Iteration 5701 / 7640) loss: 1.846247
(Epoch 15 / 20) train acc: 0.437000; val_acc: 0.447000
(Iteration 5801 / 7640) loss: 1.700906
(Iteration 5901 / 7640) loss: 2.042712
(Iteration 6001 / 7640) loss: 1.717460
(Iteration 6101 / 7640) loss: 1.720652
(Epoch 16 / 20) train acc: 0.429000; val_acc: 0.449000
(Iteration 6201 / 7640) loss: 1.771262
(Iteration 6301 / 7640) loss: 1.735663
(Iteration 6401 / 7640) loss: 1.863828
(Epoch 17 / 20) train acc: 0.484000; val_acc: 0.484000
```

```
(Iteration 6501 / 7640) loss: 1.666792
(Iteration 6601 / 7640) loss: 1.700066
(Iteration 6701 / 7640) loss: 1.670127
(Iteration 6801 / 7640) loss: 1.901586
(Epoch 18 / 20) train acc: 0.457000; val_acc: 0.443000
(Iteration 6901 / 7640) loss: 1.724731
(Iteration 7001 / 7640) loss: 1.570364
(Iteration 7101 / 7640) loss: 1.614314
(Iteration 7201 / 7640) loss: 1.674197
(Epoch 19 / 20) train acc: 0.467000; val_acc: 0.481000
(Iteration 7301 / 7640) loss: 1.689777
(Iteration 7401 / 7640) loss: 1.762023
(Iteration 7501 / 7640) loss: 1.742079
(Iteration 7601 / 7640) loss: 1.617849
(Epoch 20 / 20) train acc: 0.498000; val_acc: 0.481000
Validation accuracy: 0.479
Best validation accuracy: 0.504
Best hyperparameter configuration: {'learning_rate': 0.0005, 'weight_scale':
0.05, 'reg': 0.1}
Fine-tuning the best model…
(Iteration 1 / 19100) loss: 1.505557
(Epoch 0 / 50) train acc: 0.455000; val_acc: 0.419000
(Iteration 101 / 19100) loss: 1.878435
(Iteration 201 / 19100) loss: 1.815134
(Iteration 301 / 19100) loss: 1.954366
(Epoch 1 / 50) train acc: 0.444000; val_acc: 0.464000
(Iteration 401 / 19100) loss: 1.795447
(Iteration 501 / 19100) loss: 1.791162
(Iteration 601 / 19100) loss: 1.710666
(Iteration 701 / 19100) loss: 1.790715
(Epoch 2 / 50) train acc: 0.515000; val_acc: 0.456000
(Iteration 801 / 19100) loss: 1.778913
(Iteration 901 / 19100) loss: 1.901556
(Iteration 1001 / 19100) loss: 1.879199
(Iteration 1101 / 19100) loss: 1.809650
(Epoch 3 / 50) train acc: 0.493000; val_acc: 0.478000
(Iteration 1201 / 19100) loss: 1.749364
(Iteration 1301 / 19100) loss: 1.754424
(Iteration 1401 / 19100) loss: 1.911546
(Iteration 1501 / 19100) loss: 1.669556
(Epoch 4 / 50) train acc: 0.471000; val_acc: 0.490000
(Iteration 1601 / 19100) loss: 1.880944
(Iteration 1701 / 19100) loss: 1.587439
(Iteration 1801 / 19100) loss: 1.735412
(Iteration 1901 / 19100) loss: 1.658157
(Epoch 5 / 50) train acc: 0.509000; val_acc: 0.506000
(Iteration 2001 / 19100) loss: 1.592510
(Iteration 2101 / 19100) loss: 1.621179
```

```
(Iteration 2201 / 19100) loss: 1.550763
(Epoch 6 / 50) train acc: 0.536000; val_acc: 0.513000
(Iteration 2301 / 19100) loss: 1.603066
(Iteration 2401 / 19100) loss: 1.423574
(Iteration 2501 / 19100) loss: 1.536093
(Iteration 2601 / 19100) loss: 1.606193
(Epoch 7 / 50) train acc: 0.517000; val_acc: 0.486000
(Iteration 2701 / 19100) loss: 1.695622
(Iteration 2801 / 19100) loss: 1.524343
(Iteration 2901 / 19100) loss: 1.518763
(Iteration 3001 / 19100) loss: 1.813553
(Epoch 8 / 50) train acc: 0.533000; val_acc: 0.477000
(Iteration 3101 / 19100) loss: 1.548020
(Iteration 3201 / 19100) loss: 1.546636
(Iteration 3301 / 19100) loss: 1.499405
(Iteration 3401 / 19100) loss: 1.574035
(Epoch 9 / 50) train acc: 0.569000; val_acc: 0.517000
(Iteration 3501 / 19100) loss: 1.498258
(Iteration 3601 / 19100) loss: 1.612831
(Iteration 3701 / 19100) loss: 1.464074
(Iteration 3801 / 19100) loss: 1.735745
(Epoch 10 / 50) train acc: 0.523000; val_acc: 0.516000
(Iteration 3901 / 19100) loss: 1.461816
(Iteration 4001 / 19100) loss: 1.500769
(Iteration 4101 / 19100) loss: 1.592626
(Iteration 4201 / 19100) loss: 1.566544
(Epoch 11 / 50) train acc: 0.551000; val_acc: 0.504000
(Iteration 4301 / 19100) loss: 1.461337
(Iteration 4401 / 19100) loss: 1.567383
(Iteration 4501 / 19100) loss: 1.507856
(Epoch 12 / 50) train acc: 0.554000; val_acc: 0.510000
(Iteration 4601 / 19100) loss: 1.476897
(Iteration 4701 / 19100) loss: 1.372770
(Iteration 4801 / 19100) loss: 1.402949
(Iteration 4901 / 19100) loss: 1.569751
(Epoch 13 / 50) train acc: 0.583000; val_acc: 0.542000
(Iteration 5001 / 19100) loss: 1.302561
(Iteration 5101 / 19100) loss: 1.404372
(Iteration 5201 / 19100) loss: 1.427856
(Iteration 5301 / 19100) loss: 1.503830
(Epoch 14 / 50) train acc: 0.598000; val_acc: 0.539000
(Iteration 5401 / 19100) loss: 1.386995
(Iteration 5501 / 19100) loss: 1.346526
(Iteration 5601 / 19100) loss: 1.386125
(Iteration 5701 / 19100) loss: 1.351637
(Epoch 15 / 50) train acc: 0.565000; val_acc: 0.532000
(Iteration 5801 / 19100) loss: 1.285181
(Iteration 5901 / 19100) loss: 1.309657
```

```
(Iteration 6001 / 19100) loss: 1.357075
(Iteration 6101 / 19100) loss: 1.213683
(Epoch 16 / 50) train acc: 0.604000; val_acc: 0.535000
(Iteration 6201 / 19100) loss: 1.297855
(Iteration 6301 / 19100) loss: 1.380463
(Iteration 6401 / 19100) loss: 1.326887
(Epoch 17 / 50) train acc: 0.589000; val_acc: 0.534000
(Iteration 6501 / 19100) loss: 1.392538
(Iteration 6601 / 19100) loss: 1.149276
(Iteration 6701 / 19100) loss: 1.458917
(Iteration 6801 / 19100) loss: 1.299282
(Epoch 18 / 50) train acc: 0.632000; val_acc: 0.545000
(Iteration 6901 / 19100) loss: 1.381598
(Iteration 7001 / 19100) loss: 1.486535
(Iteration 7101 / 19100) loss: 1.377787
(Iteration 7201 / 19100) loss: 1.388935
(Epoch 19 / 50) train acc: 0.656000; val_acc: 0.551000
(Iteration 7301 / 19100) loss: 1.346175
(Iteration 7401 / 19100) loss: 1.182242
(Iteration 7501 / 19100) loss: 1.251348
(Iteration 7601 / 19100) loss: 1.194824
(Epoch 20 / 50) train acc: 0.647000; val_acc: 0.552000
(Iteration 7701 / 19100) loss: 1.285966
(Iteration 7801 / 19100) loss: 1.240621
(Iteration 7901 / 19100) loss: 1.257059
(Iteration 8001 / 19100) loss: 1.322218
(Epoch 21 / 50) train acc: 0.636000; val_acc: 0.548000
(Iteration 8101 / 19100) loss: 1.277414
(Iteration 8201 / 19100) loss: 1.184067
(Iteration 8301 / 19100) loss: 1.258924
(Iteration 8401 / 19100) loss: 1.242566
(Epoch 22 / 50) train acc: 0.667000; val_acc: 0.560000
(Iteration 8501 / 19100) loss: 1.195508
(Iteration 8601 / 19100) loss: 1.213251
(Iteration 8701 / 19100) loss: 1.087032
(Epoch 23 / 50) train acc: 0.674000; val_acc: 0.572000
(Iteration 8801 / 19100) loss: 1.042664
(Iteration 8901 / 19100) loss: 1.289443
(Iteration 9001 / 19100) loss: 1.226833
(Iteration 9101 / 19100) loss: 1.340669
(Epoch 24 / 50) train acc: 0.670000; val_acc: 0.570000
(Iteration 9201 / 19100) loss: 1.110571
(Iteration 9301 / 19100) loss: 1.142775
(Iteration 9401 / 19100) loss: 1.160328
(Iteration 9501 / 19100) loss: 1.140032
(Epoch 25 / 50) train acc: 0.692000; val_acc: 0.554000
(Iteration 9601 / 19100) loss: 1.031474
(Iteration 9701 / 19100) loss: 1.105103
```

```
(Iteration 9801 / 19100) loss: 1.292787
(Iteration 9901 / 19100) loss: 1.010416
(Epoch 26 / 50) train acc: 0.706000; val_acc: 0.556000
(Iteration 10001 / 19100) loss: 1.189206
(Iteration 10101 / 19100) loss: 1.089936
(Iteration 10201 / 19100) loss: 1.116365
(Iteration 10301 / 19100) loss: 1.107196
(Epoch 27 / 50) train acc: 0.708000; val_acc: 0.557000
(Iteration 10401 / 19100) loss: 1.173949
(Iteration 10501 / 19100) loss: 1.218692
(Iteration 10601 / 19100) loss: 1.163887
(Epoch 28 / 50) train acc: 0.747000; val_acc: 0.542000
(Iteration 10701 / 19100) loss: 1.024833
(Iteration 10801 / 19100) loss: 0.995011
(Iteration 10901 / 19100) loss: 1.040614
(Iteration 11001 / 19100) loss: 1.061909
(Epoch 29 / 50) train acc: 0.736000; val_acc: 0.557000
(Iteration 11101 / 19100) loss: 1.025043
(Iteration 11201 / 19100) loss: 0.951952
(Iteration 11301 / 19100) loss: 0.962118
(Iteration 11401 / 19100) loss: 1.008774
(Epoch 30 / 50) train acc: 0.748000; val_acc: 0.555000
(Iteration 11501 / 19100) loss: 1.052085
(Iteration 11601 / 19100) loss: 0.979444
(Iteration 11701 / 19100) loss: 0.978464
(Iteration 11801 / 19100) loss: 0.832433
(Epoch 31 / 50) train acc: 0.780000; val_acc: 0.562000
(Iteration 11901 / 19100) loss: 1.056872
(Iteration 12001 / 19100) loss: 0.968589
(Iteration 12101 / 19100) loss: 0.944655
(Iteration 12201 / 19100) loss: 0.987716
(Epoch 32 / 50) train acc: 0.777000; val_acc: 0.566000
(Iteration 12301 / 19100) loss: 0.891478
(Iteration 12401 / 19100) loss: 0.882502
(Iteration 12501 / 19100) loss: 0.871573
(Iteration 12601 / 19100) loss: 1.065584
(Epoch 33 / 50) train acc: 0.769000; val_acc: 0.556000
(Iteration 12701 / 19100) loss: 0.967124
(Iteration 12801 / 19100) loss: 0.983855
(Iteration 12901 / 19100) loss: 0.870819
(Epoch 34 / 50) train acc: 0.782000; val_acc: 0.557000
(Iteration 13001 / 19100) loss: 0.944917
(Iteration 13101 / 19100) loss: 0.953439
(Iteration 13201 / 19100) loss: 0.906536
(Iteration 13301 / 19100) loss: 1.001700
(Epoch 35 / 50) train acc: 0.804000; val_acc: 0.548000
(Iteration 13401 / 19100) loss: 0.901425
(Iteration 13501 / 19100) loss: 0.996360
```

```
(Iteration 13601 / 19100) loss: 0.994212
(Iteration 13701 / 19100) loss: 0.776022
(Epoch 36 / 50) train acc: 0.783000; val_acc: 0.567000
(Iteration 13801 / 19100) loss: 0.830546
(Iteration 13901 / 19100) loss: 0.859939
(Iteration 14001 / 19100) loss: 0.970149
(Iteration 14101 / 19100) loss: 0.850106
(Epoch 37 / 50) train acc: 0.821000; val_acc: 0.547000
(Iteration 14201 / 19100) loss: 0.890524
(Iteration 14301 / 19100) loss: 0.827415
(Iteration 14401 / 19100) loss: 0.935662
(Iteration 14501 / 19100) loss: 1.017854
(Epoch 38 / 50) train acc: 0.824000; val_acc: 0.549000
(Iteration 14601 / 19100) loss: 0.630977
(Iteration 14701 / 19100) loss: 0.802837
(Iteration 14801 / 19100) loss: 0.909213
(Epoch 39 / 50) train acc: 0.853000; val_acc: 0.556000
(Iteration 14901 / 19100) loss: 0.849287
(Iteration 15001 / 19100) loss: 0.914108
(Iteration 15101 / 19100) loss: 0.802600
(Iteration 15201 / 19100) loss: 0.789657
(Epoch 40 / 50) train acc: 0.831000; val_acc: 0.540000
(Iteration 15301 / 19100) loss: 0.796613
(Iteration 15401 / 19100) loss: 0.855191
(Iteration 15501 / 19100) loss: 0.875999
(Iteration 15601 / 19100) loss: 1.029028
(Epoch 41 / 50) train acc: 0.851000; val_acc: 0.540000
(Iteration 15701 / 19100) loss: 0.723491
(Iteration 15801 / 19100) loss: 0.865220
(Iteration 15901 / 19100) loss: 0.764570
(Iteration 16001 / 19100) loss: 0.737709
(Epoch 42 / 50) train acc: 0.856000; val_acc: 0.535000
(Iteration 16101 / 19100) loss: 0.739672
(Iteration 16201 / 19100) loss: 0.674430
(Iteration 16301 / 19100) loss: 0.830617
(Iteration 16401 / 19100) loss: 0.745429
(Epoch 43 / 50) train acc: 0.854000; val_acc: 0.535000
(Iteration 16501 / 19100) loss: 0.755620
(Iteration 16601 / 19100) loss: 0.728001
(Iteration 16701 / 19100) loss: 0.686801
(Iteration 16801 / 19100) loss: 0.732313
(Epoch 44 / 50) train acc: 0.871000; val_acc: 0.545000
(Iteration 16901 / 19100) loss: 0.706886
(Iteration 17001 / 19100) loss: 0.768484
(Iteration 17101 / 19100) loss: 0.623815
(Epoch 45 / 50) train acc: 0.891000; val_acc: 0.540000
(Iteration 17201 / 19100) loss: 0.651794
(Iteration 17301 / 19100) loss: 0.708617
```

```
(Iteration 17401 / 19100) loss: 0.717639
(Iteration 17501 / 19100) loss: 0.693816
(Epoch 46 / 50) train acc: 0.881000; val_acc: 0.537000
(Iteration 17601 / 19100) loss: 0.612721
(Iteration 17701 / 19100) loss: 0.728546
(Iteration 17801 / 19100) loss: 0.747459
(Iteration 17901 / 19100) loss: 0.731957
(Epoch 47 / 50) train acc: 0.891000; val_acc: 0.533000
(Iteration 18001 / 19100) loss: 0.688144
(Iteration 18101 / 19100) loss: 0.824459
(Iteration 18201 / 19100) loss: 0.762262
(Iteration 18301 / 19100) loss: 0.828766
(Epoch 48 / 50) train acc: 0.902000; val_acc: 0.535000
(Iteration 18401 / 19100) loss: 0.649396
(Iteration 18501 / 19100) loss: 0.655767
(Iteration 18601 / 19100) loss: 0.619167
(Iteration 18701 / 19100) loss: 0.795581
(Epoch 49 / 50) train acc: 0.906000; val_acc: 0.530000
(Iteration 18801 / 19100) loss: 0.630091
(Iteration 18901 / 19100) loss: 0.647227
(Iteration 19001 / 19100) loss: 0.590793
(Epoch 50 / 50) train acc: 0.915000; val_acc: 0.527000
Final Training accuracy: 0.394265306122449
Final Validation accuracy: 0.349
```

14 Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

[22]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-22-bc662ae98aae> in <cell line: 1>()
----> 1 y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
      2 y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
      3 print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean()
      4 print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())

KeyError: 'X_test'
```