two_layer_net

October 25, 2024

[ ]:
```
# 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. 'cse493g1/assignments/assignment2/'
FOLDERNAME = 'cse493g1/assignment2/'
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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignment2/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignment2
```

1 Fully-Connected Neural Nets

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

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

```
  out = # the output

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

  return out, cache
```

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

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

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

  return dx, dw
```

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

[ ]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cse493g1.classifiers.fc_net import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.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))))
```

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

[ ]:
```
# 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: forward

Open the file `cse493g1/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(493)
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:  2.7815615012337633e-10
dw error:  4.28112143997314e-11
db error:  7.32931845803195e-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:

[ ]:
```
np.random.seed(493)
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:  3.275625790132643e-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:

Both Sigmoid and ReLU

Sigmoid– Sigmoid has a problem with getting zero gradient when the weights are a big enough to saturate the function. Where the output is the tailend of the function (0 or 1). For example lets say we use the one dimensional case $[150, -10]$ the resulting values for sigmoid prime would be:

$$
\begin{aligned}
f(t) &= \frac{1}{1+e^{-t}} \\
f'(t) &= f(t)(1-f(t)) \\
f'(10) &= 0.000045395807736 \\
f'(150) &= 7.1750959732 \times 10^{-66}
\end{aligned}
$$

Which as we can see is quite close to zero and would kill any gradient flow backwards.

ReLu – The problem with ReLu as since its 0 whenever the input is less or equal to 0 then that means that the gradient when the input is $ <= 0$ is also 0. So an example input that could lead to zero gradient flow back would be negative values for example, $[-5, -10]$.

Leaky ReLu – Tries to solve the problem with ReLu with adding a small slope to the values less then 0. This makes it so the gradient of negative inputs would be that small slope instead of 0.

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 `cse493g1/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 cse493g1.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(493)
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:  4.791696167224978e-10
dw error:  1.877217636859801e-10
db error:  8.009352854936581e-12
```

7 Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cse493g1/layers.py`. These should be similar to what you implemented in `cse493g1/classifiers/softmax.py` and `cse493g1/classifiers/linear_svm.py` in Assignment 1.

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

[ ]:
```
np.random.seed(493)
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 sum_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:  8.998158338429793
dx error:  8.182894472887002e-10

Testing softmax_loss:
loss:  2.3024013710141706
dx error:  7.452229549289443e-09
```

8 Two-layer network

Open the file `cse493g1/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. Read through it to make sure you understand the API. You can run the cell below to test your implementation.

[ ]:
```
np.random.seed(493)
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: 3.31e-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

Open the file `cse493g1/solver.py` and read through it to familiarize yourself with the API. Additionally, familiarize yourself with the `sgd` function in `cse493g1/optim.py`. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

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

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

```
solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-4,
                    },
                    lr_decay=0.95,
                    num_epochs=5, batch_size=200,
                    print_every=100)
solver.train()

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

```
(Iteration 1 / 1225) loss: 2.300129
(Epoch 0 / 5) train acc: 0.138000; val_acc: 0.140000
(Iteration 101 / 1225) loss: 2.251986
(Iteration 201 / 1225) loss: 2.194247
(Epoch 1 / 5) train acc: 0.248000; val_acc: 0.255000
(Iteration 301 / 1225) loss: 2.100231
(Iteration 401 / 1225) loss: 2.047014
(Epoch 2 / 5) train acc: 0.310000; val_acc: 0.302000
(Iteration 501 / 1225) loss: 1.931189
(Iteration 601 / 1225) loss: 1.934842
(Iteration 701 / 1225) loss: 1.901075
(Epoch 3 / 5) train acc: 0.326000; val_acc: 0.326000
(Iteration 801 / 1225) loss: 1.818142
(Iteration 901 / 1225) loss: 1.979605
(Epoch 4 / 5) train acc: 0.345000; val_acc: 0.350000
(Iteration 1001 / 1225) loss: 1.887235
(Iteration 1101 / 1225) loss: 1.779911
(Iteration 1201 / 1225) loss: 1.723447
(Epoch 5 / 5) train acc: 0.347000; val_acc: 0.367000
```

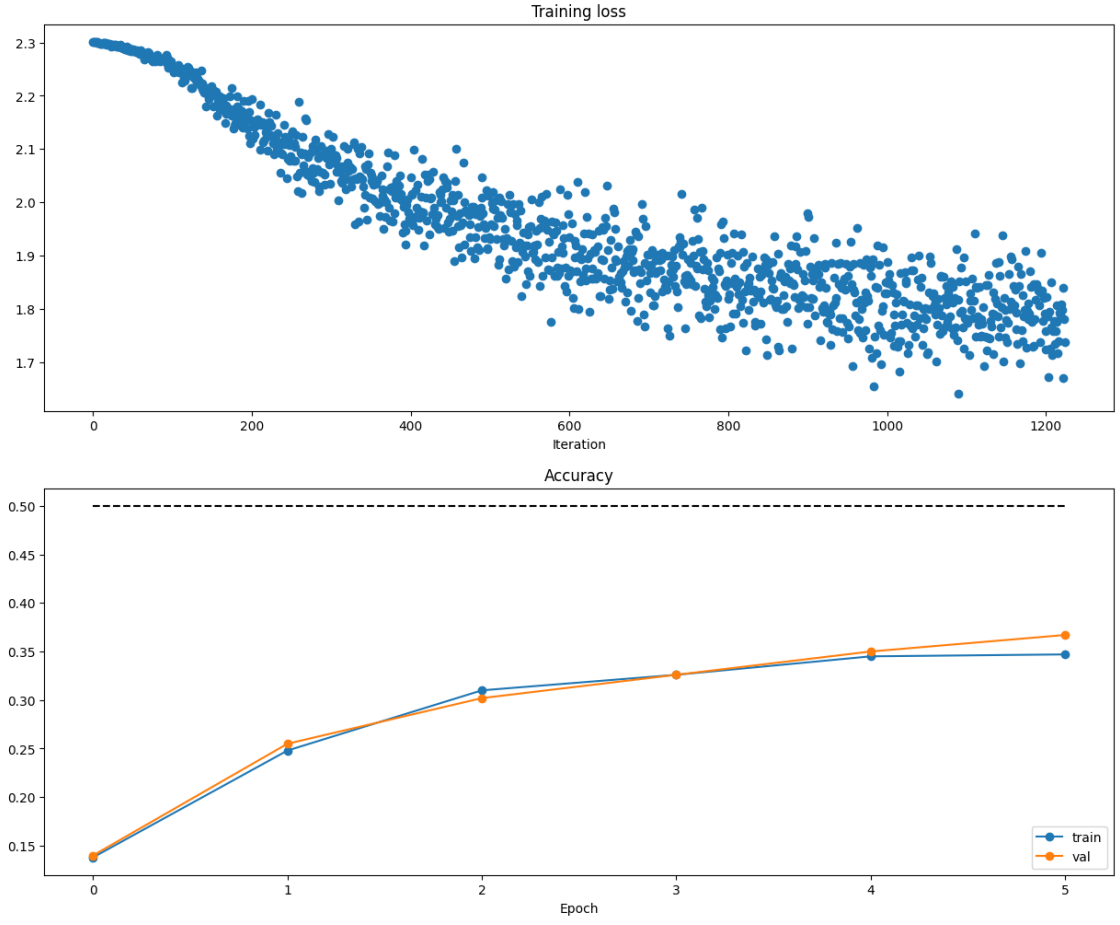
10 Debug the training

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

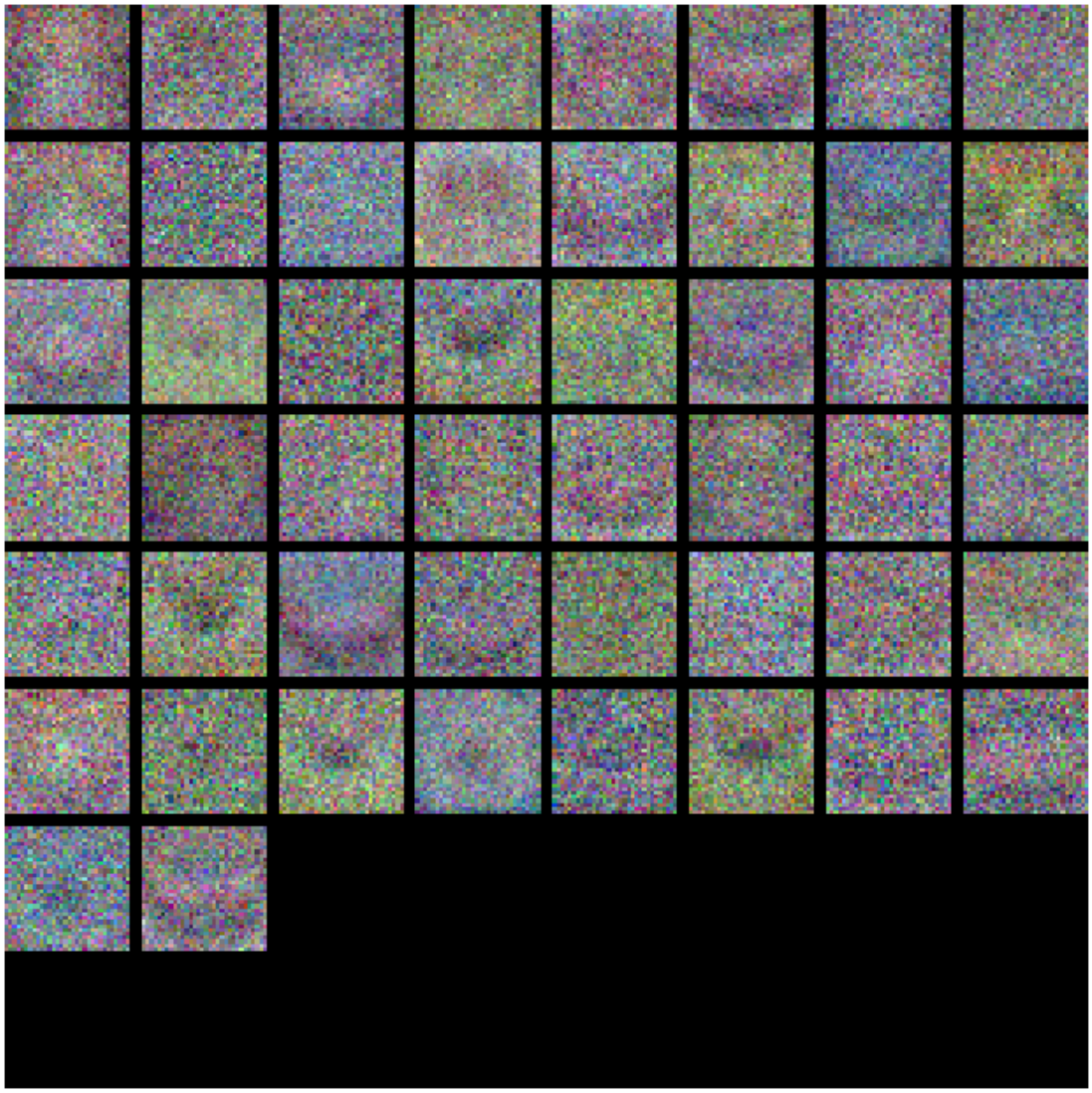
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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

[ ]:
```
# 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()
```



[ ]:
```
from cse493g1.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



11 Tune your hyperparameters

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

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

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

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

[ ]:
```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained #
# model in best_model.                                                         #
#                                                                              #
# To help debug your network, it may help to use visualizations similar to the #
# ones we used above; these visualizations will have significant qualitative   #
# differences from the ones we saw above for the poorly tuned network.         #
#                                                                              #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to #
# write code to sweep through possible combinations of hyperparameters         #
# automatically like we did on thexs previous exercises.                       #
#   #
```

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

input_size = 32 * 32 * 3
hidden_size = 75
num_classes = 10
weight_scale = 1e-3
reg = 0.48
model = TwoLayerNet(input_size, hidden_size, num_classes, weight_scale, reg)
solver = None

solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1.5e-3,
                    },
                    lr_decay=0.94,
                    num_epochs=10, batch_size=200,
                    print_every=100)
solver.train()
best_model = model


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

```
(Iteration 1 / 2450) loss: 2.356685
(Epoch 0 / 10) train acc: 0.168000; val_acc: 0.176000
(Iteration 101 / 2450) loss: 1.821846
(Iteration 201 / 2450) loss: 1.643672
(Epoch 1 / 10) train acc: 0.426000; val_acc: 0.430000
(Iteration 301 / 2450) loss: 1.659047
(Iteration 401 / 2450) loss: 1.673621
(Epoch 2 / 10) train acc: 0.457000; val_acc: 0.451000
(Iteration 501 / 2450) loss: 1.587218
(Iteration 601 / 2450) loss: 1.475716
(Iteration 701 / 2450) loss: 1.586855
(Epoch 3 / 10) train acc: 0.476000; val_acc: 0.456000
(Iteration 801 / 2450) loss: 1.525818
(Iteration 901 / 2450) loss: 1.587900
(Epoch 4 / 10) train acc: 0.506000; val_acc: 0.476000
(Iteration 1001 / 2450) loss: 1.669561
(Iteration 1101 / 2450) loss: 1.561524
(Iteration 1201 / 2450) loss: 1.464459
(Epoch 5 / 10) train acc: 0.512000; val_acc: 0.484000
```

```
(Iteration 1301 / 2450) loss: 1.534861
(Iteration 1401 / 2450) loss: 1.578719
(Epoch 6 / 10) train acc: 0.525000; val_acc: 0.499000
(Iteration 1501 / 2450) loss: 1.518960
(Iteration 1601 / 2450) loss: 1.511987
(Iteration 1701 / 2450) loss: 1.378031
(Epoch 7 / 10) train acc: 0.543000; val_acc: 0.508000
(Iteration 1801 / 2450) loss: 1.341113
(Iteration 1901 / 2450) loss: 1.531071
(Epoch 8 / 10) train acc: 0.546000; val_acc: 0.511000
(Iteration 2001 / 2450) loss: 1.594078
(Iteration 2101 / 2450) loss: 1.511680
(Iteration 2201 / 2450) loss: 1.425473
(Epoch 9 / 10) train acc: 0.590000; val_acc: 0.506000
(Iteration 2301 / 2450) loss: 1.418234
(Iteration 2401 / 2450) loss: 1.433724
(Epoch 10 / 10) train acc: 0.558000; val_acc: 0.507000
```

12 Test your model!

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

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

```
Validation set accuracy:  0.511
```

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

```
Test set accuracy:  0.511
```

12.1 Inline Question 2:

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

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

YourAnswer :

1, 3

YourExplanation :

The problem with training accracy being much higher than testing accuracy is that the model has overfit to the training data i.e. memorized the data. What the model needs to fix that it has to be more general.

Train on a larger dataset: *True* – This is because a large gap in training and testing accuracy could mean there isn't enough data in the train set to generalize to the test set. So using more data could help us lower that difference as the NN would be able to generalize better if the new data is different from the current train set. The flip side is that if the larger dataset isn't diverse enough it could lead to the same problem.

Add more hidden units: *False* – This would just add to the difference as this would add capacity to the model to allow for more overfitting on the trainset. Since we want to fix the delta in accuracies this would not be a solution. More complexity/capacity would lead to more memorization of the train set, sufficient regularization could fix that but thats the next option.

Increase the regularization strength: *True* – We can assume in a sense that the model is complex enough since the trian accuracy in this senario is higher than the test accuaracy. So with regularization it will reduce memorization of the data and allow for the model to classify data it hasn't seen better. As the NN would be more genral as a classifier reducing the delta between the accuracies.

features

October 25, 2024

[12]:
```
# 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. 'cse493g1/assignments/assignment2/'
FOLDERNAME = 'cse493g1/assignment2/'
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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignment2/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignment2
```

1 Image features exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

[13]:
```
import random
import numpy as np
from cse493g1.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%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 extenrnal modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

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

1.1 Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

[14]:
```
from cse493g1.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cse493g1/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may␣
 ↪cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

```
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

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

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

1.2 Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your own interest.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

[15]:
```
from cse493g1.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img,␣
 ↪nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
```

```
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

```
Done extracting features for 1000 / 49000 images
Done extracting features for 2000 / 49000 images
Done extracting features for 3000 / 49000 images
Done extracting features for 4000 / 49000 images
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
```

```
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

1.3 Train SVM on features

Using the multiclass SVM code developed in Assignment 1 (copy and paste your code from Assignment 1 into the file `cse493g1/classifiers/linear_svm.py`), train SVMs on top of the features extracted above. This should achieve better results than training SVMs directly on top of raw pixels.

[16]:
```
# Use the validation set to tune the learning rate and regularization strength

from cse493g1.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-7, 1e-2, 1e-6]
regularization_strengths = [3, 1e-3, 1e-4]

results = {}
best_val = -1
best_svm = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.43 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

grid_search = [ (lr,reg) for lr in learning_rates for reg in␣
 ↪regularization_strengths ]

for lr, reg in grid_search:
  svm = LinearSVM()
  loss_hist = svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,␣
 ↪num_iters=2000)
  y_train_pred = svm.predict(X_train_feats)
  train_accuracy = np.mean(y_train == y_train_pred)
  y_val_pred = svm.predict(X_val_feats)
  val_accuracy = np.mean(y_val == y_val_pred)
  results[(lr,reg)] = (train_accuracy, val_accuracy)
  if val_accuracy > best_val:
    best_val = val_accuracy
```

```
    best_svm = svm

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

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved: %f' % best_val)
```

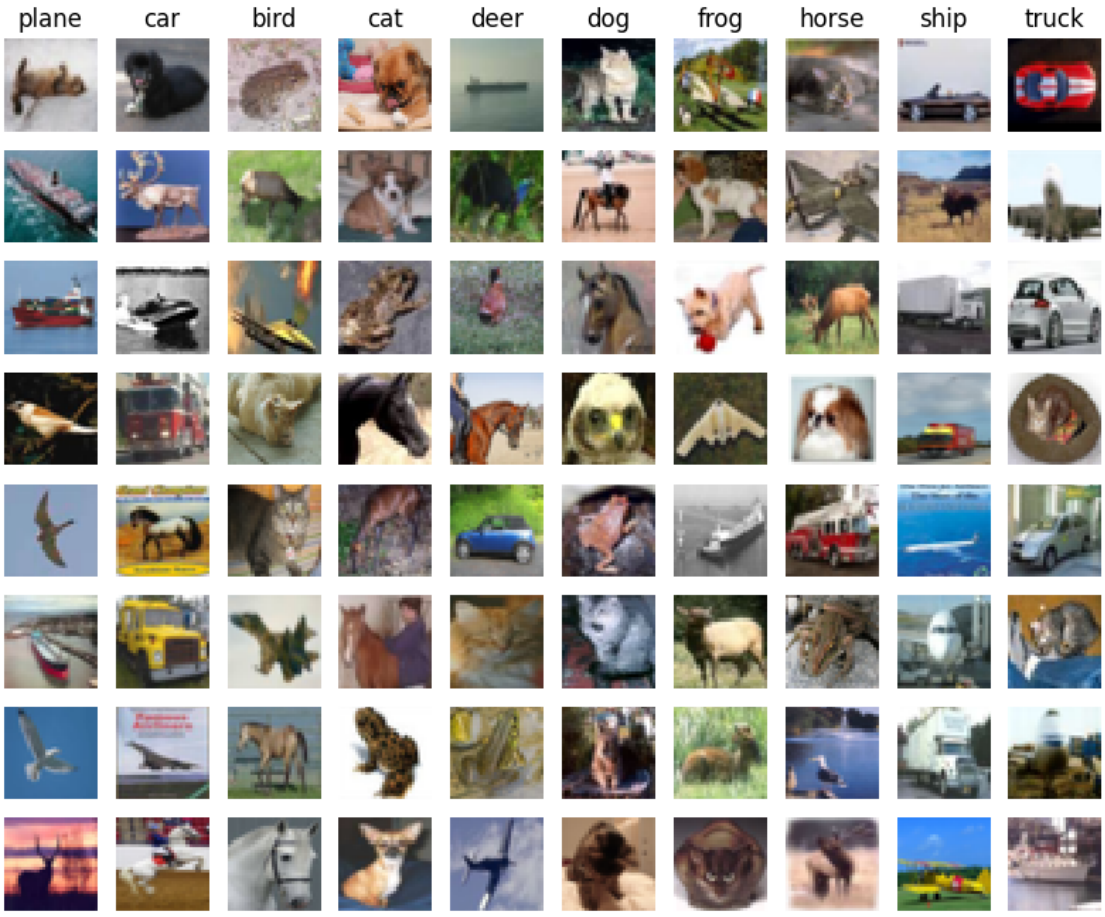
```
lr 1.000000e-07 reg 1.000000e-04 train accuracy: 0.121224 val accuracy: 0.121000
lr 1.000000e-07 reg 1.000000e-03 train accuracy: 0.111143 val accuracy: 0.118000
lr 1.000000e-07 reg 3.000000e+00 train accuracy: 0.097286 val accuracy: 0.103000
lr 1.000000e-06 reg 1.000000e-04 train accuracy: 0.275286 val accuracy: 0.268000
lr 1.000000e-06 reg 1.000000e-03 train accuracy: 0.288122 val accuracy: 0.283000
lr 1.000000e-06 reg 3.000000e+00 train accuracy: 0.304857 val accuracy: 0.312000
lr 1.000000e-02 reg 1.000000e-04 train accuracy: 0.511429 val accuracy: 0.506000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.510367 val accuracy: 0.503000
lr 1.000000e-02 reg 3.000000e+00 train accuracy: 0.456980 val accuracy: 0.469000
best validation accuracy achieved: 0.506000
```

[17]:
```
# Evaluate your trained SVM on the test set: you should be able to get at least␣
 ↪0.40
y_test_pred = best_svm.predict(X_test_feats)
test_accuracy = np.mean(y_test == y_test_pred)
print(test_accuracy)
```

```
0.488
```

[18]:
```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls +␣
 ↪1)
        plt.imshow(X_test[idx].astype('uint8'))
```

```
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



1.3.1 Inline question 1:

Describe the misclassification results that you see. Do they make sense?

Your Answer :

There are some misclassifications that are understandable such as the animals which were mismatched pretty oftem amongst each other.

The ones I find intresting are backgrounds being the key identifier for the mismatch.For example in the imgaes labled as a plane we can see that many of them have a grey-ish blue-ish background, which makes sense as planes are usually photographed along with the sky.

Then with the ship its a similar story where I would assume that pictures of ships are taken near the sea quite often and we also see that many of the missclassifications have either water-like bodies

in the image.

Then for the other missclasifications we could say that some of them could be due the feature / color seperation as some classes could have more of certain color like white with planes.

1.4 Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

[19]:
```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 155)
(49000, 154)
```

[20]:
```
from cse493g1.classifiers.fc_net import TwoLayerNet
from cse493g1.solver import Solver

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

#net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
```

```
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_val = 0
weight_scale = 5e-2
for i in range(10):
  lr = 10**np.random.uniform(-1.5, 0)
  reg = 10**np.random.uniform(-7, -4)
  hidden_dim = np.random.randint(100, 500)
  net = TwoLayerNet(input_dim, hidden_dim, num_classes, weight_scale, reg)
  solver = Solver(net, data,
                      update_rule='sgd',
                      optim_config={
                        'learning_rate': lr,
                      },
                      lr_decay=0.95,
                      num_epochs=12, batch_size=200, print_every=100,
                      verbose = False)
  solver.train()

  y_val_pred = np.argmax(net.loss(data['X_val']), axis=1)
  val_acc = (y_val_pred == data['y_val']).mean()
  if i == 0 or best_val < val_acc:
    best_net = net
    best_val = val_acc

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

[21]:
```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

y_test_pred = np.argmax(best_net.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.575
```

FullyConnectedNets

October 25, 2024

[6]:
```
# 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. 'cse493g1/assignments/assignment2/'
FOLDERNAME = 'cse493g1/assignment2/'
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/$FOLDERNAME/cse493g1/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/cse493g1/assignment2/cse493g1/datasets
/content/drive/My Drive/cse493g1/assignment2
```

1 Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cse493g1/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assignment, you will be implementing layers in `cse493g1/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from the previous notebook.

[7]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cse493g1.classifiers.fc_net import *
from cse493g1.data_utils import get_CIFAR10_data
from cse493g1.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cse493g1.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"

%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))))
```

[8]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{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,)
```

1.1 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. This is a good way to see if the initial losses seem reasonable.

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

[9]:
```
np.random.seed(493)
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(f"{name} relative error: {rel_error(grad_num, grads[name])}")
```

```
Running check with reg =  0
Initial loss:  2.299821914918452
W1 relative error: 9.656953386228676e-08
W2 relative error: 4.42914245511279e-06
W3 relative error: 1.4112116043233687e-06
b1 relative error: 9.10105007110364e-09
b2 relative error: 1.3597116499374134e-07
b3 relative error: 1.3432054300218008e-10
Running check with reg =  3.14
Initial loss:  6.991659719884911
W1 relative error: 2.2442521861888318e-08
W2 relative error: 1.3072726683559537e-08
W3 relative error: 1.585488123924185e-08
b1 relative error: 4.368076841105884e-08
b2 relative error: 1.0110675182132624e-08
b3 relative error: 1.1311009016240723e-10
```

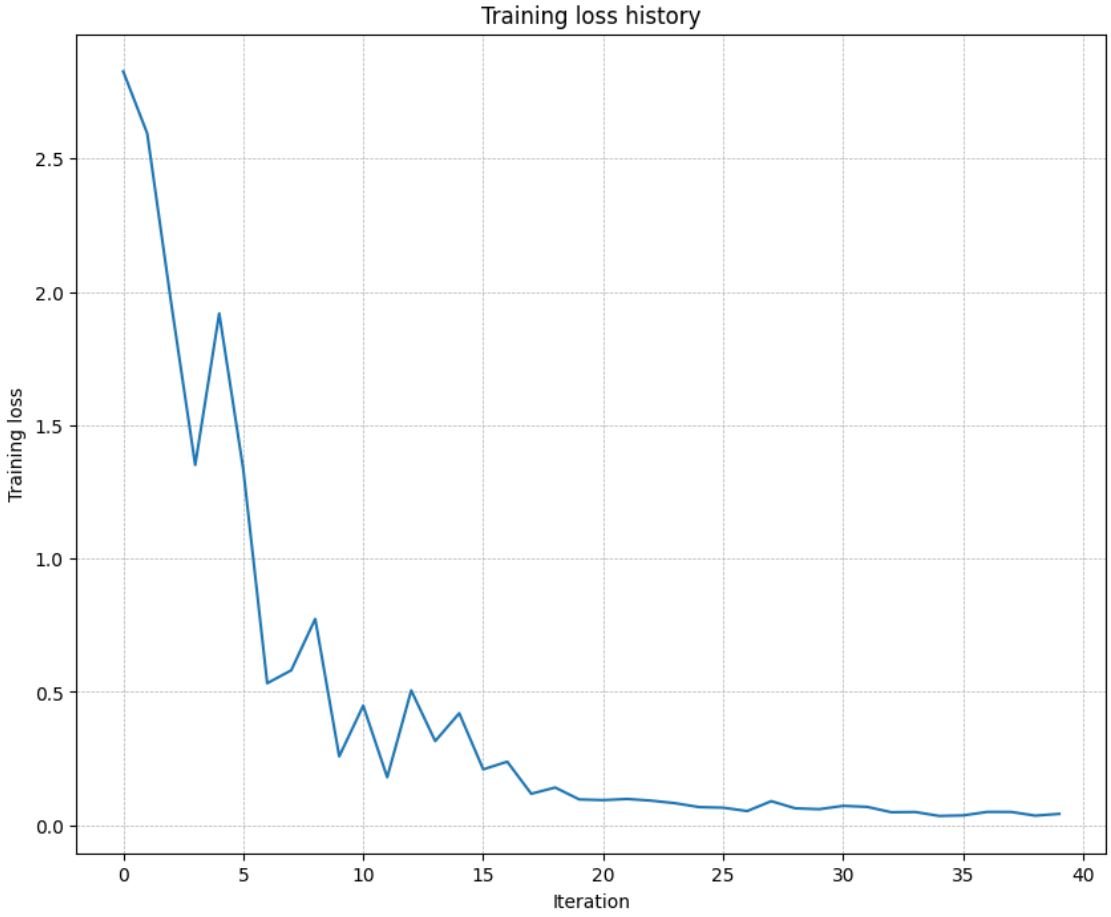
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy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

[10]:
```
# 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],
  "X_val": data["X_val"],
  "y_val": data["y_val"],
}

weight_scale = 2.2e-2   # Experiment with this!
learning_rate = 3.7e-3  # Experiment with this!
model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 2.827078
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.131000
(Epoch 1 / 20) train acc: 0.420000; val_acc: 0.149000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.137000
(Epoch 3 / 20) train acc: 0.820000; val_acc: 0.145000
(Epoch 4 / 20) train acc: 0.880000; val_acc: 0.166000
(Epoch 5 / 20) train acc: 0.900000; val_acc: 0.161000
(Iteration 11 / 40) loss: 0.448723
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.162000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.164000
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.168000
(Iteration 21 / 40) loss: 0.094574
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.163000
```

```
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.168000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.164000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.166000
(Iteration 31 / 40) loss: 0.073026
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.162000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.160000
```



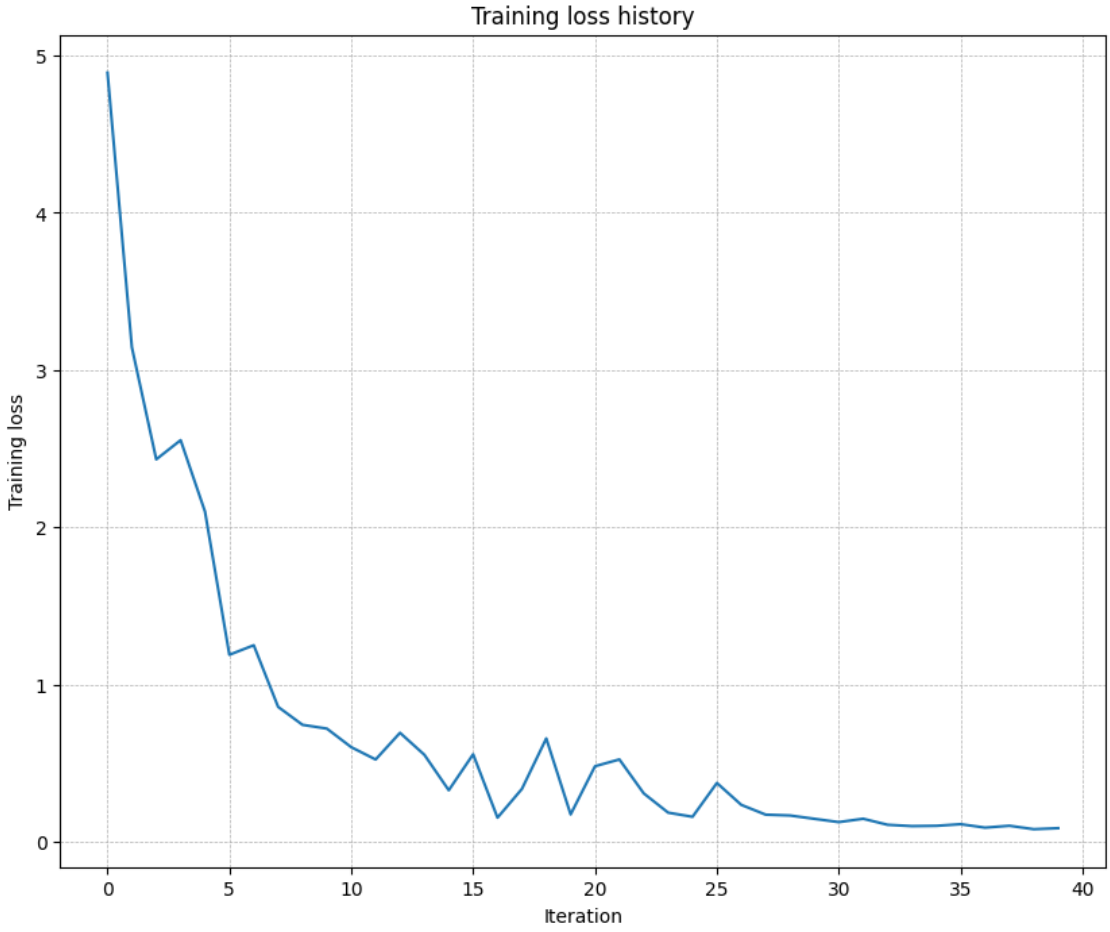
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[11]:
```
# 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'],
}

learning_rate = 1.8e-3  # Experiment with this!
weight_scale = 5.7e-2   # Experiment with this!
model = FullyConnectedNet(
    [100, 100, 100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule='sgd',
    optim_config={'learning_rate': learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 4.890899
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.115000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.119000
(Epoch 2 / 20) train acc: 0.540000; val_acc: 0.114000
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.124000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.106000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.108000
(Iteration 11 / 40) loss: 0.601747
(Epoch 6 / 20) train acc: 0.860000; val_acc: 0.116000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.115000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.131000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.127000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.125000
(Iteration 21 / 40) loss: 0.480634
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.118000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.121000
```

```
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.120000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.119000
(Iteration 31 / 40) loss: 0.125297
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.119000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.124000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.125000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.125000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.127000
```



1.2 Inline Question 1:

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

1.3 Answer:

The five-layer network was more sensitve to the initialization scale. This is do to the multiplicitive property of the chain rule. As due to larger initializations that would result in larger values for our gradients and as those large values multiply (even more so due to increased layers) the gradient values will balloon. The same idea can apply if the intialized values are too small as then the small values will result in the same idea but the gradients will deflate and go towards zero. This is present less so in the three-layer network since there is less chain rule (less layers).

2 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.

2.1 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. See the Momentum Update section for more information.

Open the file `cse493g1/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.

[12]:
```
from cse493g1.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
```

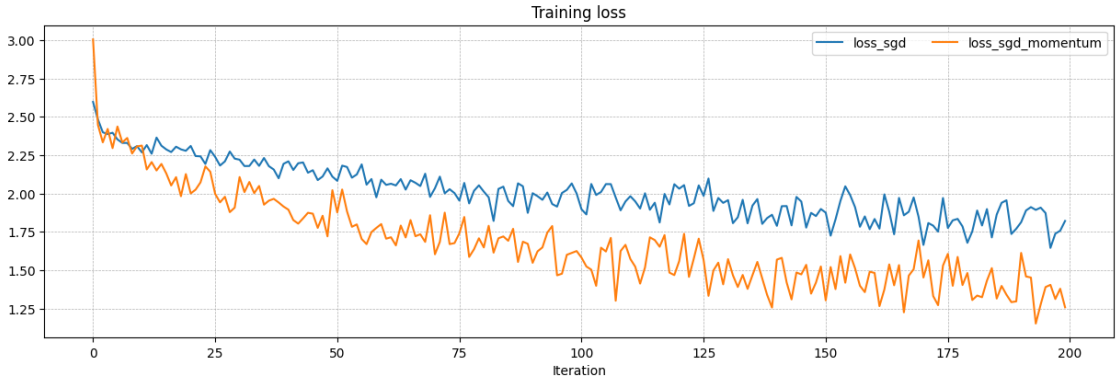
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

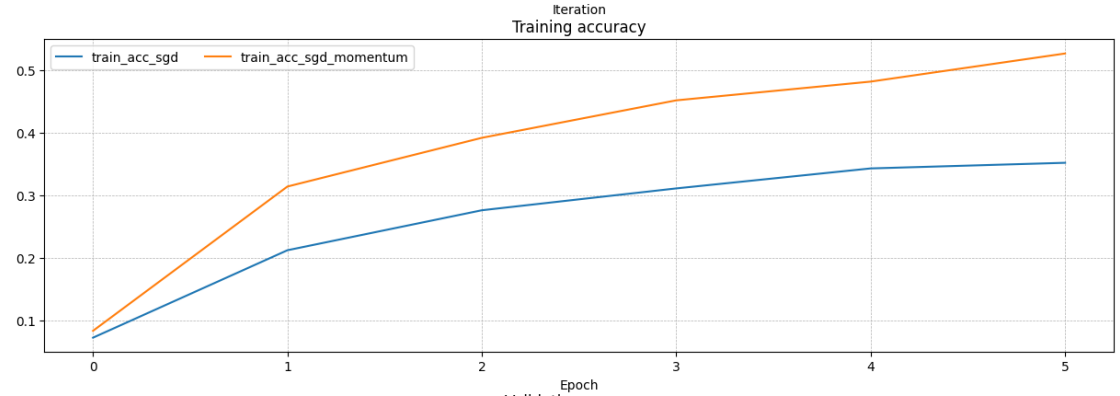
[13]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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': 5e-3},
        verbose=True,
    )
    solvers[update_rule] = solver
    solver.train()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"loss_{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"train_acc_{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"val_acc_{update_rule}")
```

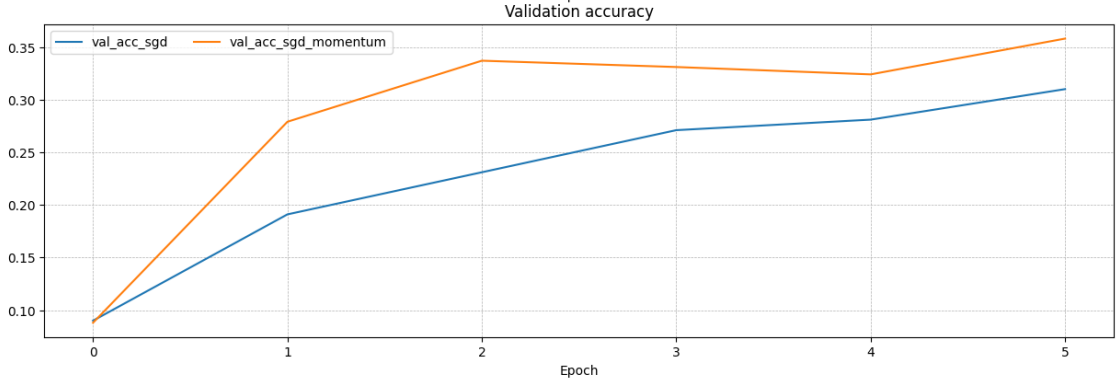
```python
for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.596947
(Epoch 0 / 5) train acc: 0.072000; val_acc: 0.090000
(Iteration 11 / 200) loss: 2.268104
(Iteration 21 / 200) loss: 2.310026
(Iteration 31 / 200) loss: 2.220483
(Epoch 1 / 5) train acc: 0.212000; val_acc: 0.191000
(Iteration 41 / 200) loss: 2.209802
(Iteration 51 / 200) loss: 2.082704
(Iteration 61 / 200) loss: 2.056447
(Iteration 71 / 200) loss: 2.034617
(Epoch 2 / 5) train acc: 0.276000; val_acc: 0.231000
(Iteration 81 / 200) loss: 2.012472
(Iteration 91 / 200) loss: 2.001965
(Iteration 101 / 200) loss: 1.897417
(Iteration 111 / 200) loss: 1.982470
(Epoch 3 / 5) train acc: 0.311000; val_acc: 0.271000
(Iteration 121 / 200) loss: 2.031500
(Iteration 131 / 200) loss: 1.956905
(Iteration 141 / 200) loss: 1.789398
(Iteration 151 / 200) loss: 1.875251
(Epoch 4 / 5) train acc: 0.343000; val_acc: 0.281000
(Iteration 161 / 200) loss: 1.834020
(Iteration 171 / 200) loss: 1.664942
(Iteration 181 / 200) loss: 1.753409
(Iteration 191 / 200) loss: 1.812975
(Epoch 5 / 5) train acc: 0.352000; val_acc: 0.310000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 3.004274
(Epoch 0 / 5) train acc: 0.083000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.312269
(Iteration 21 / 200) loss: 2.000611
(Iteration 31 / 200) loss: 2.107353
(Epoch 1 / 5) train acc: 0.314000; val_acc: 0.279000
(Iteration 41 / 200) loss: 1.894668
(Iteration 51 / 200) loss: 1.876688
(Iteration 61 / 200) loss: 1.705844
(Iteration 71 / 200) loss: 1.604111
(Epoch 2 / 5) train acc: 0.392000; val_acc: 0.337000
(Iteration 81 / 200) loss: 1.648795
```

```
(Iteration 91 / 200) loss: 1.548430
(Iteration 101 / 200) loss: 1.582610
(Iteration 111 / 200) loss: 1.573521
(Epoch 3 / 5) train acc: 0.452000; val_acc: 0.331000
(Iteration 121 / 200) loss: 1.559532
(Iteration 131 / 200) loss: 1.572553
(Iteration 141 / 200) loss: 1.569282
(Iteration 151 / 200) loss: 1.303874
(Epoch 4 / 5) train acc: 0.482000; val_acc: 0.324000
(Iteration 161 / 200) loss: 1.482469
(Iteration 171 / 200) loss: 1.452708
(Iteration 181 / 200) loss: 1.306036
(Iteration 191 / 200) loss: 1.612928
(Epoch 5 / 5) train acc: 0.527000; val_acc: 0.358000
```







2.2 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cse493g1/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

[14]:
```
# Test RMSProp implementation
from cse493g1.optim import rmsprop

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)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

[15]:
```
# Test Adam implementation
from cse493g1.optim import adam
```

```
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)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

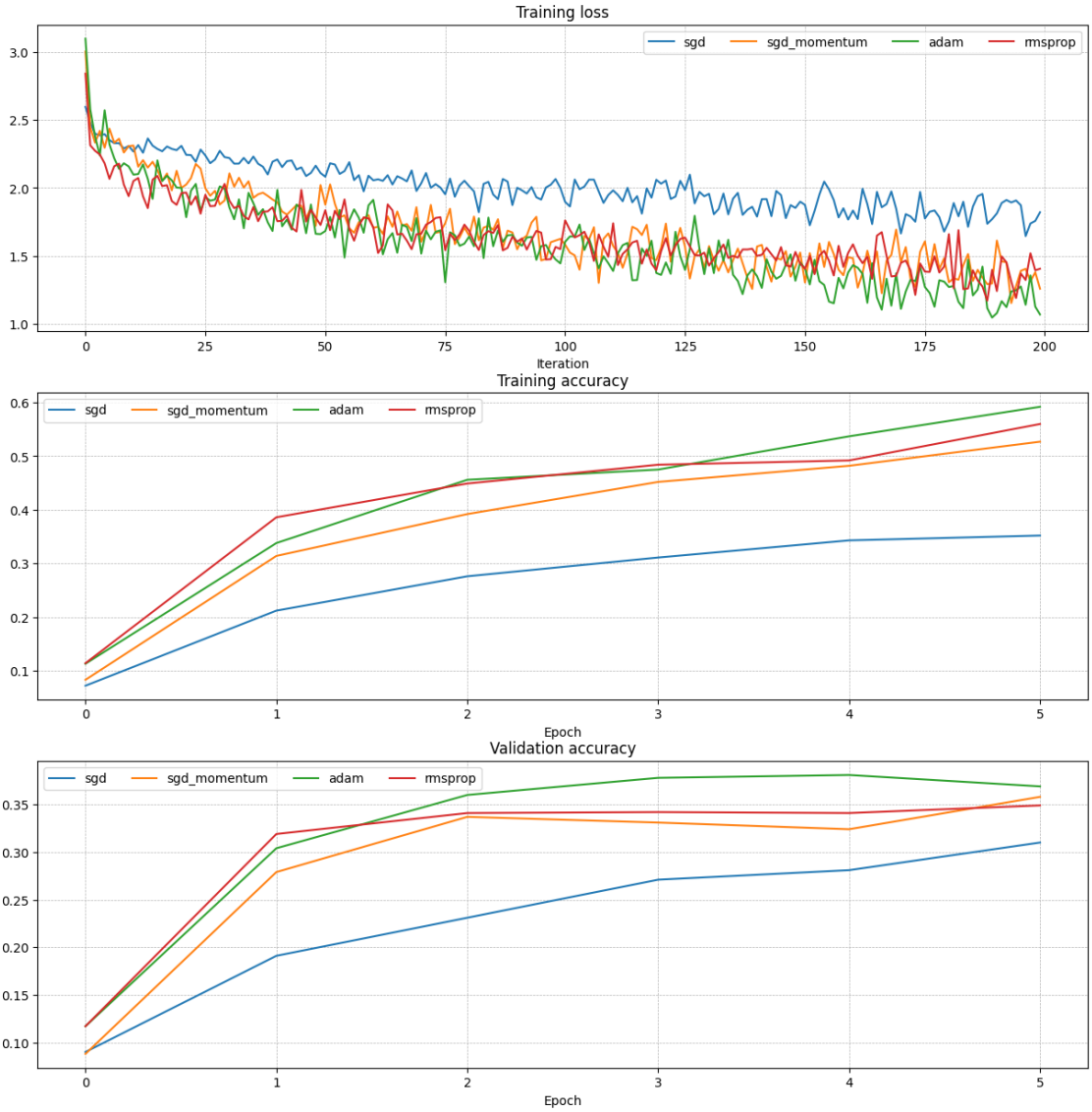
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[16]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
    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': learning_rates[update_rule]},
        verbose=True
    )
    solvers[update_rule] = solver
    solver.train()
    print()

fig, axes = plt.subplots(3, 1, figsize=(15, 15))

axes[0].set_title('Training loss')
axes[0].set_xlabel('Iteration')
axes[1].set_title('Training accuracy')
axes[1].set_xlabel('Epoch')
axes[2].set_title('Validation accuracy')
axes[2].set_xlabel('Epoch')

for update_rule, solver in solvers.items():
    axes[0].plot(solver.loss_history, label=f"{update_rule}")
    axes[1].plot(solver.train_acc_history, label=f"{update_rule}")
    axes[2].plot(solver.val_acc_history, label=f"{update_rule}")

for ax in axes:
    ax.legend(loc='best', ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  adam
(Iteration 1 / 200) loss: 3.098972
(Epoch 0 / 5) train acc: 0.113000; val_acc: 0.117000
(Iteration 11 / 200) loss: 2.097885
(Iteration 21 / 200) loss: 2.000393
(Iteration 31 / 200) loss: 1.856791
(Epoch 1 / 5) train acc: 0.338000; val_acc: 0.304000
(Iteration 41 / 200) loss: 1.986063
(Iteration 51 / 200) loss: 1.682858
(Iteration 61 / 200) loss: 1.912636
(Iteration 71 / 200) loss: 1.516613
(Epoch 2 / 5) train acc: 0.456000; val_acc: 0.360000
(Iteration 81 / 200) loss: 1.642534
(Iteration 91 / 200) loss: 1.598813
(Iteration 101 / 200) loss: 1.603976
(Iteration 111 / 200) loss: 1.389070
(Epoch 3 / 5) train acc: 0.475000; val_acc: 0.378000
```

```
(Iteration 121 / 200) loss: 1.360125
(Iteration 131 / 200) loss: 1.557555
(Iteration 141 / 200) loss: 1.351395
(Iteration 151 / 200) loss: 1.407298
(Epoch 4 / 5) train acc: 0.537000; val_acc: 0.381000
(Iteration 161 / 200) loss: 1.431740
(Iteration 171 / 200) loss: 1.110009
(Iteration 181 / 200) loss: 1.271121
(Iteration 191 / 200) loss: 1.079040
(Epoch 5 / 5) train acc: 0.592000; val_acc: 0.369000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.841640
(Epoch 0 / 5) train acc: 0.114000; val_acc: 0.117000
(Iteration 11 / 200) loss: 2.051360
(Iteration 21 / 200) loss: 1.962443
(Iteration 31 / 200) loss: 1.906751
(Epoch 1 / 5) train acc: 0.386000; val_acc: 0.319000
(Iteration 41 / 200) loss: 1.755408
(Iteration 51 / 200) loss: 1.836533
(Iteration 61 / 200) loss: 1.711051
(Iteration 71 / 200) loss: 1.663315
(Epoch 2 / 5) train acc: 0.449000; val_acc: 0.341000
(Iteration 81 / 200) loss: 1.684352
(Iteration 91 / 200) loss: 1.604322
(Iteration 101 / 200) loss: 1.761576
(Iteration 111 / 200) loss: 1.740179
(Epoch 3 / 5) train acc: 0.484000; val_acc: 0.342000
(Iteration 121 / 200) loss: 1.534499
(Iteration 131 / 200) loss: 1.428315
(Iteration 141 / 200) loss: 1.500756
(Iteration 151 / 200) loss: 1.405584
(Epoch 4 / 5) train acc: 0.492000; val_acc: 0.341000
(Iteration 161 / 200) loss: 1.585194
(Iteration 171 / 200) loss: 1.448813
(Iteration 181 / 200) loss: 1.660250
(Iteration 191 / 200) loss: 1.241025
(Epoch 5 / 5) train acc: 0.560000; val_acc: 0.349000
```



2.3 Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

2.4 Answer:

The reason that AdaGrad causes the network to learn slowly as time goes on is because of the historical summing of all the gradients squared. This causes the "cache" to grow larger and larger, representing this with limits we get

$lim_{t $\rightarrow\infty$} $ cache $ $= + \infty$$

then

$lim_{t $\rightarrow\infty$} $ 1 / cache $ $= 0$$

So over time the learning rate is being multiplied by values closer and closer to 0 resulting in slow learning.

Then Adam doesnt have this issue because instead of summing all previous squared gradients it uses more of a momentum approach where it keeps some small amount (decay) of the previous "cache" then just uses the current gradient squared for the rest (1- decay) of the new cache. This is based from the RMSProp (or leaky AdaGrad) where the previous squared gradient "leaks" into the next rather than being summed together.

3 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 network.

[24]:
```
best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. Store your␣
 ↪best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
input_dim = data['X_train'].shape[1]
num_classes = 10

best_val = 0
for i in range(10):
  lr = 10**np.random.uniform(-5, -2)
  reg = 10**np.random.uniform(-3, -1.5)
  weight_scale = 10**np.random.uniform(-2, -1)
  H1, H2 = 100, 100
  print(f"lr: {lr}, reg: {reg}, weight_scale: {weight_scale}")
  net = FullyConnectedNet([H1, H2], reg=reg ,weight_scale=weight_scale,
    dtype=np.float64)

  solver = Solver(net, data,
                      update_rule='adam',
```

```
                      optim_config={
                        'learning_rate': lr,
                      },
                      lr_decay=0.95,
                      num_epochs=15, batch_size=200, print_every=100,
                      verbose = True)
  solver.train()

  y_val_pred = np.argmax(net.loss(data['X_val']), axis=1)
  val_acc = (y_val_pred == data['y_val']).mean()
  if i == 0 or best_val < val_acc:
    best_model = net
    best_val = val_acc

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

```
lr: 0.0001215662819447897, reg: 0.011905549869686437, weight_scale:
0.03662918712719861
(Iteration 1 / 3675) loss: 18.953307
(Epoch 0 / 15) train acc: 0.096000; val_acc: 0.117000
(Iteration 101 / 3675) loss: 5.830413
(Iteration 201 / 3675) loss: 4.724559
(Epoch 1 / 15) train acc: 0.325000; val_acc: 0.323000
(Iteration 301 / 3675) loss: 4.571307
(Iteration 401 / 3675) loss: 4.381137
(Epoch 2 / 15) train acc: 0.414000; val_acc: 0.350000
(Iteration 501 / 3675) loss: 4.193893
(Iteration 601 / 3675) loss: 4.048876
(Iteration 701 / 3675) loss: 3.904583
(Epoch 3 / 15) train acc: 0.472000; val_acc: 0.376000
(Iteration 801 / 3675) loss: 3.891879
(Iteration 901 / 3675) loss: 3.867893
(Epoch 4 / 15) train acc: 0.441000; val_acc: 0.399000
(Iteration 1001 / 3675) loss: 3.716016
(Iteration 1101 / 3675) loss: 3.587863
(Iteration 1201 / 3675) loss: 3.656614
(Epoch 5 / 15) train acc: 0.494000; val_acc: 0.421000
(Iteration 1301 / 3675) loss: 3.388599
(Iteration 1401 / 3675) loss: 3.522232
(Epoch 6 / 15) train acc: 0.498000; val_acc: 0.430000
(Iteration 1501 / 3675) loss: 3.377637
(Iteration 1601 / 3675) loss: 3.281435
(Iteration 1701 / 3675) loss: 3.183174
(Epoch 7 / 15) train acc: 0.539000; val_acc: 0.437000
```

```
(Iteration 1801 / 3675) loss: 3.400724
(Iteration 1901 / 3675) loss: 3.363931
(Epoch 8 / 15) train acc: 0.541000; val_acc: 0.450000
(Iteration 2001 / 3675) loss: 3.047736
(Iteration 2101 / 3675) loss: 3.146546
(Iteration 2201 / 3675) loss: 3.004704
(Epoch 9 / 15) train acc: 0.536000; val_acc: 0.441000
(Iteration 2301 / 3675) loss: 2.866244
(Iteration 2401 / 3675) loss: 3.003234
(Epoch 10 / 15) train acc: 0.548000; val_acc: 0.449000
(Iteration 2501 / 3675) loss: 3.052377
(Iteration 2601 / 3675) loss: 2.927915
(Epoch 11 / 15) train acc: 0.560000; val_acc: 0.449000
(Iteration 2701 / 3675) loss: 2.704002
(Iteration 2801 / 3675) loss: 2.770071
(Iteration 2901 / 3675) loss: 2.889073
(Epoch 12 / 15) train acc: 0.556000; val_acc: 0.457000
(Iteration 3001 / 3675) loss: 2.762889
(Iteration 3101 / 3675) loss: 2.640046
(Epoch 13 / 15) train acc: 0.603000; val_acc: 0.468000
(Iteration 3201 / 3675) loss: 2.585809
(Iteration 3301 / 3675) loss: 2.589494
(Iteration 3401 / 3675) loss: 2.704568
(Epoch 14 / 15) train acc: 0.581000; val_acc: 0.464000
(Iteration 3501 / 3675) loss: 2.424904
(Iteration 3601 / 3675) loss: 2.655728
(Epoch 15 / 15) train acc: 0.626000; val_acc: 0.467000
lr: 0.0005893715283857147, reg: 0.026362033279860423, weight_scale:
0.018694328453814565
(Iteration 1 / 3675) loss: 4.130349
(Epoch 0 / 15) train acc: 0.182000; val_acc: 0.192000
(Iteration 101 / 3675) loss: 2.581647
(Iteration 201 / 3675) loss: 2.153568
(Epoch 1 / 15) train acc: 0.447000; val_acc: 0.450000
(Iteration 301 / 3675) loss: 2.059406
(Iteration 401 / 3675) loss: 1.691700
(Epoch 2 / 15) train acc: 0.473000; val_acc: 0.442000
(Iteration 501 / 3675) loss: 1.601347
(Iteration 601 / 3675) loss: 1.709392
(Iteration 701 / 3675) loss: 1.534188
(Epoch 3 / 15) train acc: 0.479000; val_acc: 0.480000
(Iteration 801 / 3675) loss: 1.551054
(Iteration 901 / 3675) loss: 1.571403
(Epoch 4 / 15) train acc: 0.514000; val_acc: 0.480000
(Iteration 1001 / 3675) loss: 1.553482
(Iteration 1101 / 3675) loss: 1.397949
(Iteration 1201 / 3675) loss: 1.658737
(Epoch 5 / 15) train acc: 0.538000; val_acc: 0.475000
```

```
(Iteration 1301 / 3675) loss: 1.483773
(Iteration 1401 / 3675) loss: 1.536008
(Epoch 6 / 15) train acc: 0.508000; val_acc: 0.486000
(Iteration 1501 / 3675) loss: 1.583072
(Iteration 1601 / 3675) loss: 1.443771
(Iteration 1701 / 3675) loss: 1.331112
(Epoch 7 / 15) train acc: 0.508000; val_acc: 0.488000
(Iteration 1801 / 3675) loss: 1.466295
(Iteration 1901 / 3675) loss: 1.238032
(Epoch 8 / 15) train acc: 0.513000; val_acc: 0.499000
(Iteration 2001 / 3675) loss: 1.444992
(Iteration 2101 / 3675) loss: 1.454366
(Iteration 2201 / 3675) loss: 1.443077
(Epoch 9 / 15) train acc: 0.561000; val_acc: 0.494000
(Iteration 2301 / 3675) loss: 1.381595
(Iteration 2401 / 3675) loss: 1.430891
(Epoch 10 / 15) train acc: 0.532000; val_acc: 0.492000
(Iteration 2501 / 3675) loss: 1.448707
(Iteration 2601 / 3675) loss: 1.530786
(Epoch 11 / 15) train acc: 0.596000; val_acc: 0.522000
(Iteration 2701 / 3675) loss: 1.430934
(Iteration 2801 / 3675) loss: 1.382526
(Iteration 2901 / 3675) loss: 1.340484
(Epoch 12 / 15) train acc: 0.601000; val_acc: 0.535000
(Iteration 3001 / 3675) loss: 1.454013
(Iteration 3101 / 3675) loss: 1.389422
(Epoch 13 / 15) train acc: 0.590000; val_acc: 0.498000
(Iteration 3201 / 3675) loss: 1.464160
(Iteration 3301 / 3675) loss: 1.480211
(Iteration 3401 / 3675) loss: 1.303220
(Epoch 14 / 15) train acc: 0.565000; val_acc: 0.515000
(Iteration 3501 / 3675) loss: 1.362303
(Iteration 3601 / 3675) loss: 1.365951
(Epoch 15 / 15) train acc: 0.561000; val_acc: 0.540000
lr: 0.0007701675206042822, reg: 0.005284499748175352, weight_scale:
0.012017361322504187
(Iteration 1 / 3675) loss: 2.444434
(Epoch 0 / 15) train acc: 0.183000; val_acc: 0.187000
(Iteration 101 / 3675) loss: 1.686479
(Iteration 201 / 3675) loss: 1.621009
(Epoch 1 / 15) train acc: 0.462000; val_acc: 0.451000
(Iteration 301 / 3675) loss: 1.509525
(Iteration 401 / 3675) loss: 1.560431
(Epoch 2 / 15) train acc: 0.503000; val_acc: 0.462000
(Iteration 501 / 3675) loss: 1.588460
(Iteration 601 / 3675) loss: 1.494736
(Iteration 701 / 3675) loss: 1.442943
(Epoch 3 / 15) train acc: 0.508000; val_acc: 0.477000
```

```
(Iteration 801 / 3675) loss: 1.473004
(Iteration 901 / 3675) loss: 1.468869
(Epoch 4 / 15) train acc: 0.539000; val_acc: 0.460000
(Iteration 1001 / 3675) loss: 1.318982
(Iteration 1101 / 3675) loss: 1.400469
(Iteration 1201 / 3675) loss: 1.530310
(Epoch 5 / 15) train acc: 0.520000; val_acc: 0.490000
(Iteration 1301 / 3675) loss: 1.449743
(Iteration 1401 / 3675) loss: 1.411316
(Epoch 6 / 15) train acc: 0.589000; val_acc: 0.512000
(Iteration 1501 / 3675) loss: 1.468320
(Iteration 1601 / 3675) loss: 1.402214
(Iteration 1701 / 3675) loss: 1.312454
(Epoch 7 / 15) train acc: 0.571000; val_acc: 0.505000
(Iteration 1801 / 3675) loss: 1.448455
(Iteration 1901 / 3675) loss: 1.309005
(Epoch 8 / 15) train acc: 0.574000; val_acc: 0.530000
(Iteration 2001 / 3675) loss: 1.162997
(Iteration 2101 / 3675) loss: 1.228776
(Iteration 2201 / 3675) loss: 1.259191
(Epoch 9 / 15) train acc: 0.608000; val_acc: 0.540000
(Iteration 2301 / 3675) loss: 1.405739
(Iteration 2401 / 3675) loss: 1.428299
(Epoch 10 / 15) train acc: 0.583000; val_acc: 0.535000
(Iteration 2501 / 3675) loss: 1.228752
(Iteration 2601 / 3675) loss: 1.295771
(Epoch 11 / 15) train acc: 0.612000; val_acc: 0.544000
(Iteration 2701 / 3675) loss: 1.270628
(Iteration 2801 / 3675) loss: 1.287559
(Iteration 2901 / 3675) loss: 1.276372
(Epoch 12 / 15) train acc: 0.623000; val_acc: 0.511000
(Iteration 3001 / 3675) loss: 1.174541
(Iteration 3101 / 3675) loss: 1.241550
(Epoch 13 / 15) train acc: 0.598000; val_acc: 0.521000
(Iteration 3201 / 3675) loss: 1.226902
(Iteration 3301 / 3675) loss: 1.224686
(Iteration 3401 / 3675) loss: 1.076229
(Epoch 14 / 15) train acc: 0.642000; val_acc: 0.543000
(Iteration 3501 / 3675) loss: 1.178754
(Iteration 3601 / 3675) loss: 1.130259
(Epoch 15 / 15) train acc: 0.632000; val_acc: 0.529000
lr: 0.000141407114752084, reg: 0.023461391586694157, weight_scale:
0.03873215024604771
(Iteration 1 / 3675) loss: 26.220140
(Epoch 0 / 15) train acc: 0.130000; val_acc: 0.103000
(Iteration 101 / 3675) loss: 9.300604
(Iteration 201 / 3675) loss: 8.061437
(Epoch 1 / 15) train acc: 0.327000; val_acc: 0.315000
```

```
(Iteration 301 / 3675) loss: 7.368567
(Iteration 401 / 3675) loss: 7.125089
(Epoch 2 / 15) train acc: 0.380000; val_acc: 0.355000
(Iteration 501 / 3675) loss: 6.885090
(Iteration 601 / 3675) loss: 6.585286
(Iteration 701 / 3675) loss: 6.282101
(Epoch 3 / 15) train acc: 0.470000; val_acc: 0.388000
(Iteration 801 / 3675) loss: 6.095878
(Iteration 901 / 3675) loss: 6.002246
(Epoch 4 / 15) train acc: 0.469000; val_acc: 0.408000
(Iteration 1001 / 3675) loss: 5.748163
(Iteration 1101 / 3675) loss: 5.521704
(Iteration 1201 / 3675) loss: 5.531807
(Epoch 5 / 15) train acc: 0.482000; val_acc: 0.420000
(Iteration 1301 / 3675) loss: 5.162943
(Iteration 1401 / 3675) loss: 5.149166
(Epoch 6 / 15) train acc: 0.540000; val_acc: 0.443000
(Iteration 1501 / 3675) loss: 4.950170
(Iteration 1601 / 3675) loss: 4.719410
(Iteration 1701 / 3675) loss: 4.772789
(Epoch 7 / 15) train acc: 0.526000; val_acc: 0.442000
(Iteration 1801 / 3675) loss: 4.519326
(Iteration 1901 / 3675) loss: 4.383520
(Epoch 8 / 15) train acc: 0.553000; val_acc: 0.459000
(Iteration 2001 / 3675) loss: 4.464107
(Iteration 2101 / 3675) loss: 4.136182
(Iteration 2201 / 3675) loss: 4.136010
(Epoch 9 / 15) train acc: 0.552000; val_acc: 0.475000
(Iteration 2301 / 3675) loss: 3.851745
(Iteration 2401 / 3675) loss: 3.997228
(Epoch 10 / 15) train acc: 0.554000; val_acc: 0.476000
(Iteration 2501 / 3675) loss: 3.852250
(Iteration 2601 / 3675) loss: 3.616742
(Epoch 11 / 15) train acc: 0.573000; val_acc: 0.467000
(Iteration 2701 / 3675) loss: 3.675794
(Iteration 2801 / 3675) loss: 3.458310
(Iteration 2901 / 3675) loss: 3.582544
(Epoch 12 / 15) train acc: 0.560000; val_acc: 0.475000
(Iteration 3001 / 3675) loss: 3.363856
(Iteration 3101 / 3675) loss: 3.132931
(Epoch 13 / 15) train acc: 0.617000; val_acc: 0.488000
(Iteration 3201 / 3675) loss: 3.127808
(Iteration 3301 / 3675) loss: 3.195020
(Iteration 3401 / 3675) loss: 2.997448
(Epoch 14 / 15) train acc: 0.602000; val_acc: 0.489000
(Iteration 3501 / 3675) loss: 2.921829
(Iteration 3601 / 3675) loss: 2.991046
(Epoch 15 / 15) train acc: 0.635000; val_acc: 0.498000
```

```
lr: 5.2396730775860646e-05, reg: 0.0018424404684168297, weight_scale:
0.022364876088094406
(Iteration 1 / 3675) loss: 2.933340
(Epoch 0 / 15) train acc: 0.097000; val_acc: 0.111000
(Iteration 101 / 3675) loss: 2.010951
(Iteration 201 / 3675) loss: 1.930244
(Epoch 1 / 15) train acc: 0.391000; val_acc: 0.410000
(Iteration 301 / 3675) loss: 1.803214
(Iteration 401 / 3675) loss: 1.926039
(Epoch 2 / 15) train acc: 0.436000; val_acc: 0.441000
(Iteration 501 / 3675) loss: 1.832409
(Iteration 601 / 3675) loss: 1.732292
(Iteration 701 / 3675) loss: 1.683492
(Epoch 3 / 15) train acc: 0.446000; val_acc: 0.450000
(Iteration 801 / 3675) loss: 1.613011
(Iteration 901 / 3675) loss: 1.607049
(Epoch 4 / 15) train acc: 0.481000; val_acc: 0.474000
(Iteration 1001 / 3675) loss: 1.672610
(Iteration 1101 / 3675) loss: 1.654615
(Iteration 1201 / 3675) loss: 1.571079
(Epoch 5 / 15) train acc: 0.511000; val_acc: 0.479000
(Iteration 1301 / 3675) loss: 1.700190
(Iteration 1401 / 3675) loss: 1.514568
(Epoch 6 / 15) train acc: 0.537000; val_acc: 0.493000
(Iteration 1501 / 3675) loss: 1.488252
(Iteration 1601 / 3675) loss: 1.717201
(Iteration 1701 / 3675) loss: 1.629690
(Epoch 7 / 15) train acc: 0.525000; val_acc: 0.499000
(Iteration 1801 / 3675) loss: 1.491600
(Iteration 1901 / 3675) loss: 1.372765
(Epoch 8 / 15) train acc: 0.549000; val_acc: 0.501000
(Iteration 2001 / 3675) loss: 1.495744
(Iteration 2101 / 3675) loss: 1.525273
(Iteration 2201 / 3675) loss: 1.365461
(Epoch 9 / 15) train acc: 0.540000; val_acc: 0.499000
(Iteration 2301 / 3675) loss: 1.584522
(Iteration 2401 / 3675) loss: 1.491090
(Epoch 10 / 15) train acc: 0.537000; val_acc: 0.494000
(Iteration 2501 / 3675) loss: 1.441305
(Iteration 2601 / 3675) loss: 1.386982
(Epoch 11 / 15) train acc: 0.569000; val_acc: 0.502000
(Iteration 2701 / 3675) loss: 1.528629
(Iteration 2801 / 3675) loss: 1.512061
(Iteration 2901 / 3675) loss: 1.401570
(Epoch 12 / 15) train acc: 0.570000; val_acc: 0.516000
(Iteration 3001 / 3675) loss: 1.396425
(Iteration 3101 / 3675) loss: 1.314794
(Epoch 13 / 15) train acc: 0.559000; val_acc: 0.514000
```

```
(Iteration 3201 / 3675) loss: 1.504888
(Iteration 3301 / 3675) loss: 1.415838
(Iteration 3401 / 3675) loss: 1.272936
(Epoch 14 / 15) train acc: 0.566000; val_acc: 0.515000
(Iteration 3501 / 3675) loss: 1.296776
(Iteration 3601 / 3675) loss: 1.233217
(Epoch 15 / 15) train acc: 0.578000; val_acc: 0.513000
lr: 0.00018072387850545513, reg: 0.0015624235684854359, weight_scale:
0.019209960914625457
(Iteration 1 / 3675) loss: 2.718843
(Epoch 0 / 15) train acc: 0.164000; val_acc: 0.152000
(Iteration 101 / 3675) loss: 1.761054
(Iteration 201 / 3675) loss: 1.738211
(Epoch 1 / 15) train acc: 0.471000; val_acc: 0.456000
(Iteration 301 / 3675) loss: 1.500566
(Iteration 401 / 3675) loss: 1.467984
(Epoch 2 / 15) train acc: 0.505000; val_acc: 0.467000
(Iteration 501 / 3675) loss: 1.494137
(Iteration 601 / 3675) loss: 1.389827
(Iteration 701 / 3675) loss: 1.339460
(Epoch 3 / 15) train acc: 0.587000; val_acc: 0.488000
(Iteration 801 / 3675) loss: 1.272302
(Iteration 901 / 3675) loss: 1.370244
(Epoch 4 / 15) train acc: 0.542000; val_acc: 0.500000
(Iteration 1001 / 3675) loss: 1.317950
(Iteration 1101 / 3675) loss: 1.333066
(Iteration 1201 / 3675) loss: 1.325909
(Epoch 5 / 15) train acc: 0.572000; val_acc: 0.511000
(Iteration 1301 / 3675) loss: 1.220459
(Iteration 1401 / 3675) loss: 1.228222
(Epoch 6 / 15) train acc: 0.612000; val_acc: 0.513000
(Iteration 1501 / 3675) loss: 1.126551
(Iteration 1601 / 3675) loss: 1.346469
(Iteration 1701 / 3675) loss: 1.160102
(Epoch 7 / 15) train acc: 0.629000; val_acc: 0.512000
(Iteration 1801 / 3675) loss: 1.178783
(Iteration 1901 / 3675) loss: 1.217931
(Epoch 8 / 15) train acc: 0.611000; val_acc: 0.517000
(Iteration 2001 / 3675) loss: 1.282185
(Iteration 2101 / 3675) loss: 1.172402
(Iteration 2201 / 3675) loss: 1.056867
(Epoch 9 / 15) train acc: 0.629000; val_acc: 0.518000
(Iteration 2301 / 3675) loss: 1.060184
(Iteration 2401 / 3675) loss: 1.169337
(Epoch 10 / 15) train acc: 0.631000; val_acc: 0.510000
(Iteration 2501 / 3675) loss: 1.030317
(Iteration 2601 / 3675) loss: 1.147701
(Epoch 11 / 15) train acc: 0.644000; val_acc: 0.536000
```

```
(Iteration 2701 / 3675) loss: 1.024266
(Iteration 2801 / 3675) loss: 1.052548
(Iteration 2901 / 3675) loss: 1.094270
(Epoch 12 / 15) train acc: 0.680000; val_acc: 0.513000
(Iteration 3001 / 3675) loss: 1.035346
(Iteration 3101 / 3675) loss: 1.114282
(Epoch 13 / 15) train acc: 0.667000; val_acc: 0.520000
(Iteration 3201 / 3675) loss: 1.092741
(Iteration 3301 / 3675) loss: 1.117166
(Iteration 3401 / 3675) loss: 1.113903
(Epoch 14 / 15) train acc: 0.683000; val_acc: 0.520000
(Iteration 3501 / 3675) loss: 0.999269
(Iteration 3601 / 3675) loss: 0.996170
(Epoch 15 / 15) train acc: 0.687000; val_acc: 0.509000
lr: 1.1976827835568655e-05, reg: 0.01679443449140798, weight_scale:
0.04931771873497703
(Iteration 1 / 3675) loss: 34.297807
(Epoch 0 / 15) train acc: 0.090000; val_acc: 0.102000
(Iteration 101 / 3675) loss: 21.417217
(Iteration 201 / 3675) loss: 18.051928
(Epoch 1 / 15) train acc: 0.198000; val_acc: 0.179000
(Iteration 301 / 3675) loss: 16.309209
(Iteration 401 / 3675) loss: 15.700837
(Epoch 2 / 15) train acc: 0.226000; val_acc: 0.217000
(Iteration 501 / 3675) loss: 15.691311
(Iteration 601 / 3675) loss: 14.045491
(Iteration 701 / 3675) loss: 14.036250
(Epoch 3 / 15) train acc: 0.194000; val_acc: 0.223000
(Iteration 801 / 3675) loss: 14.174513
(Iteration 901 / 3675) loss: 12.918035
(Epoch 4 / 15) train acc: 0.261000; val_acc: 0.231000
(Iteration 1001 / 3675) loss: 13.051059
(Iteration 1101 / 3675) loss: 12.936180
(Iteration 1201 / 3675) loss: 11.947371
(Epoch 5 / 15) train acc: 0.218000; val_acc: 0.240000
(Iteration 1301 / 3675) loss: 11.804984
(Iteration 1401 / 3675) loss: 12.317950
(Epoch 6 / 15) train acc: 0.264000; val_acc: 0.246000
(Iteration 1501 / 3675) loss: 12.068415
(Iteration 1601 / 3675) loss: 11.206646
(Iteration 1701 / 3675) loss: 11.487507
(Epoch 7 / 15) train acc: 0.263000; val_acc: 0.259000
(Iteration 1801 / 3675) loss: 12.017110
(Iteration 1901 / 3675) loss: 10.687243
(Epoch 8 / 15) train acc: 0.245000; val_acc: 0.265000
(Iteration 2001 / 3675) loss: 11.288464
(Iteration 2101 / 3675) loss: 10.842837
(Iteration 2201 / 3675) loss: 10.632445
```

```
(Epoch 9 / 15) train acc: 0.267000; val_acc: 0.266000
(Iteration 2301 / 3675) loss: 10.825404
(Iteration 2401 / 3675) loss: 10.853363
(Epoch 10 / 15) train acc: 0.267000; val_acc: 0.267000
(Iteration 2501 / 3675) loss: 10.497422
(Iteration 2601 / 3675) loss: 10.420584
(Epoch 11 / 15) train acc: 0.270000; val_acc: 0.261000
(Iteration 2701 / 3675) loss: 10.254502
(Iteration 2801 / 3675) loss: 10.438959
(Iteration 2901 / 3675) loss: 10.342123
(Epoch 12 / 15) train acc: 0.267000; val_acc: 0.258000
(Iteration 3001 / 3675) loss: 10.154753
(Iteration 3101 / 3675) loss: 10.776746
(Epoch 13 / 15) train acc: 0.303000; val_acc: 0.261000
(Iteration 3201 / 3675) loss: 9.999667
(Iteration 3301 / 3675) loss: 9.953953
(Iteration 3401 / 3675) loss: 10.144969
(Epoch 14 / 15) train acc: 0.300000; val_acc: 0.264000
(Iteration 3501 / 3675) loss: 9.590459
(Iteration 3601 / 3675) loss: 9.659367
(Epoch 15 / 15) train acc: 0.293000; val_acc: 0.265000
lr: 0.00019710495957105814, reg: 0.019281267036001786, weight_scale:
0.048770324939899495
(Iteration 1 / 3675) loss: 52.088111
(Epoch 0 / 15) train acc: 0.088000; val_acc: 0.094000
(Iteration 101 / 3675) loss: 13.699454
(Iteration 201 / 3675) loss: 11.090285
(Epoch 1 / 15) train acc: 0.333000; val_acc: 0.296000
(Iteration 301 / 3675) loss: 10.057832
(Iteration 401 / 3675) loss: 9.482916
(Epoch 2 / 15) train acc: 0.314000; val_acc: 0.296000
(Iteration 501 / 3675) loss: 9.011008
(Iteration 601 / 3675) loss: 8.854937
(Iteration 701 / 3675) loss: 8.264097
(Epoch 3 / 15) train acc: 0.378000; val_acc: 0.331000
(Iteration 801 / 3675) loss: 7.912440
(Iteration 901 / 3675) loss: 7.691950
(Epoch 4 / 15) train acc: 0.415000; val_acc: 0.346000
(Iteration 1001 / 3675) loss: 7.636295
(Iteration 1101 / 3675) loss: 7.450349
(Iteration 1201 / 3675) loss: 7.246230
(Epoch 5 / 15) train acc: 0.430000; val_acc: 0.383000
(Iteration 1301 / 3675) loss: 7.155867
(Iteration 1401 / 3675) loss: 6.848941
(Epoch 6 / 15) train acc: 0.451000; val_acc: 0.393000
(Iteration 1501 / 3675) loss: 6.655727
(Iteration 1601 / 3675) loss: 6.440511
(Iteration 1701 / 3675) loss: 6.239934
```

```
(Epoch 7 / 15) train acc: 0.432000; val_acc: 0.419000
(Iteration 1801 / 3675) loss: 6.155554
(Iteration 1901 / 3675) loss: 5.948842
(Epoch 8 / 15) train acc: 0.515000; val_acc: 0.416000
(Iteration 2001 / 3675) loss: 5.911788
(Iteration 2101 / 3675) loss: 5.694047
(Iteration 2201 / 3675) loss: 5.510148
(Epoch 9 / 15) train acc: 0.494000; val_acc: 0.434000
(Iteration 2301 / 3675) loss: 5.344808
(Iteration 2401 / 3675) loss: 5.383457
(Epoch 10 / 15) train acc: 0.504000; val_acc: 0.439000
(Iteration 2501 / 3675) loss: 4.897075
(Iteration 2601 / 3675) loss: 4.836653
(Epoch 11 / 15) train acc: 0.530000; val_acc: 0.428000
(Iteration 2701 / 3675) loss: 4.807618
(Iteration 2801 / 3675) loss: 4.631069
(Iteration 2901 / 3675) loss: 4.678254
(Epoch 12 / 15) train acc: 0.532000; val_acc: 0.445000
(Iteration 3001 / 3675) loss: 4.479152
(Iteration 3101 / 3675) loss: 4.300257
(Epoch 13 / 15) train acc: 0.536000; val_acc: 0.462000
(Iteration 3201 / 3675) loss: 4.113329
(Iteration 3301 / 3675) loss: 4.068125
(Iteration 3401 / 3675) loss: 3.952179
(Epoch 14 / 15) train acc: 0.569000; val_acc: 0.465000
(Iteration 3501 / 3675) loss: 3.967175
(Iteration 3601 / 3675) loss: 3.896696
(Epoch 15 / 15) train acc: 0.528000; val_acc: 0.462000
lr: 0.00019810883630280872, reg: 0.0051124710979752646, weight_scale:
0.011387564370987585
(Iteration 1 / 3675) loss: 2.426710
(Epoch 0 / 15) train acc: 0.175000; val_acc: 0.181000
(Iteration 101 / 3675) loss: 1.630709
(Iteration 201 / 3675) loss: 1.751527
(Epoch 1 / 15) train acc: 0.478000; val_acc: 0.460000
(Iteration 301 / 3675) loss: 1.514911
(Iteration 401 / 3675) loss: 1.638659
(Epoch 2 / 15) train acc: 0.496000; val_acc: 0.479000
(Iteration 501 / 3675) loss: 1.431310
(Iteration 601 / 3675) loss: 1.431662
(Iteration 701 / 3675) loss: 1.389813
(Epoch 3 / 15) train acc: 0.551000; val_acc: 0.514000
(Iteration 801 / 3675) loss: 1.340422
(Iteration 901 / 3675) loss: 1.489871
(Epoch 4 / 15) train acc: 0.587000; val_acc: 0.524000
(Iteration 1001 / 3675) loss: 1.291832
(Iteration 1101 / 3675) loss: 1.258461
(Iteration 1201 / 3675) loss: 1.230671
```

```
(Epoch 5 / 15) train acc: 0.575000; val_acc: 0.513000
(Iteration 1301 / 3675) loss: 1.329091
(Iteration 1401 / 3675) loss: 1.186044
(Epoch 6 / 15) train acc: 0.604000; val_acc: 0.537000
(Iteration 1501 / 3675) loss: 1.270041
(Iteration 1601 / 3675) loss: 1.309080
(Iteration 1701 / 3675) loss: 1.344569
(Epoch 7 / 15) train acc: 0.593000; val_acc: 0.538000
(Iteration 1801 / 3675) loss: 1.058234
(Iteration 1901 / 3675) loss: 1.313534
(Epoch 8 / 15) train acc: 0.627000; val_acc: 0.534000
(Iteration 2001 / 3675) loss: 1.340913
(Iteration 2101 / 3675) loss: 1.087549
(Iteration 2201 / 3675) loss: 1.162245
(Epoch 9 / 15) train acc: 0.635000; val_acc: 0.524000
(Iteration 2301 / 3675) loss: 1.092948
(Iteration 2401 / 3675) loss: 1.137898
(Epoch 10 / 15) train acc: 0.639000; val_acc: 0.536000
(Iteration 2501 / 3675) loss: 1.192303
(Iteration 2601 / 3675) loss: 0.975421
(Epoch 11 / 15) train acc: 0.660000; val_acc: 0.531000
(Iteration 2701 / 3675) loss: 1.146498
(Iteration 2801 / 3675) loss: 1.156481
(Iteration 2901 / 3675) loss: 1.115303
(Epoch 12 / 15) train acc: 0.672000; val_acc: 0.548000
(Iteration 3001 / 3675) loss: 1.097213
(Iteration 3101 / 3675) loss: 0.957974
(Epoch 13 / 15) train acc: 0.691000; val_acc: 0.525000
(Iteration 3201 / 3675) loss: 0.962681
(Iteration 3301 / 3675) loss: 1.156014
(Iteration 3401 / 3675) loss: 0.915191
(Epoch 14 / 15) train acc: 0.704000; val_acc: 0.545000
(Iteration 3501 / 3675) loss: 1.008681
(Iteration 3601 / 3675) loss: 0.986821
(Epoch 15 / 15) train acc: 0.705000; val_acc: 0.545000
lr: 7.934919116407494e-05, reg: 0.006032570902492442, weight_scale:
0.044838708043840436
(Iteration 1 / 3675) loss: 25.176360
(Epoch 0 / 15) train acc: 0.103000; val_acc: 0.091000
(Iteration 101 / 3675) loss: 7.409190
(Iteration 201 / 3675) loss: 6.685223
(Epoch 1 / 15) train acc: 0.294000; val_acc: 0.271000
(Iteration 301 / 3675) loss: 5.857523
(Iteration 401 / 3675) loss: 5.006349
(Epoch 2 / 15) train acc: 0.315000; val_acc: 0.268000
(Iteration 501 / 3675) loss: 5.175508
(Iteration 601 / 3675) loss: 4.531174
(Iteration 701 / 3675) loss: 4.485745
```

```
(Epoch 3 / 15) train acc: 0.323000; val_acc: 0.285000
(Iteration 801 / 3675) loss: 4.396559
(Iteration 901 / 3675) loss: 3.752556
(Epoch 4 / 15) train acc: 0.368000; val_acc: 0.305000
(Iteration 1001 / 3675) loss: 3.820116
(Iteration 1101 / 3675) loss: 3.946763
(Iteration 1201 / 3675) loss: 3.709832
(Epoch 5 / 15) train acc: 0.373000; val_acc: 0.316000
(Iteration 1301 / 3675) loss: 3.629800
(Iteration 1401 / 3675) loss: 3.589724
(Epoch 6 / 15) train acc: 0.403000; val_acc: 0.322000
(Iteration 1501 / 3675) loss: 3.703147
(Iteration 1601 / 3675) loss: 3.629181
(Iteration 1701 / 3675) loss: 3.411213
(Epoch 7 / 15) train acc: 0.441000; val_acc: 0.328000
(Iteration 1801 / 3675) loss: 3.480273
(Iteration 1901 / 3675) loss: 3.379425
(Epoch 8 / 15) train acc: 0.442000; val_acc: 0.341000
(Iteration 2001 / 3675) loss: 3.307994
(Iteration 2101 / 3675) loss: 3.438199
(Iteration 2201 / 3675) loss: 3.352655
(Epoch 9 / 15) train acc: 0.436000; val_acc: 0.349000
(Iteration 2301 / 3675) loss: 3.595872
(Iteration 2401 / 3675) loss: 3.262321
(Epoch 10 / 15) train acc: 0.465000; val_acc: 0.348000
(Iteration 2501 / 3675) loss: 3.428776
(Iteration 2601 / 3675) loss: 3.206253
(Epoch 11 / 15) train acc: 0.465000; val_acc: 0.368000
(Iteration 2701 / 3675) loss: 3.249345
(Iteration 2801 / 3675) loss: 3.188374
(Iteration 2901 / 3675) loss: 3.438837
(Epoch 12 / 15) train acc: 0.477000; val_acc: 0.385000
(Iteration 3001 / 3675) loss: 3.256892
(Iteration 3101 / 3675) loss: 3.170540
(Epoch 13 / 15) train acc: 0.501000; val_acc: 0.369000
(Iteration 3201 / 3675) loss: 3.254698
(Iteration 3301 / 3675) loss: 3.116381
(Iteration 3401 / 3675) loss: 3.049275
(Epoch 14 / 15) train acc: 0.473000; val_acc: 0.381000
(Iteration 3501 / 3675) loss: 3.095696
(Iteration 3601 / 3675) loss: 3.117688
(Epoch 15 / 15) train acc: 0.500000; val_acc: 0.396000
```

4 Test Your Model!

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

[25]:
```
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())
```

```
Validation set accuracy:  0.548
Test set accuracy:  0.533
```