FullyConnectedNets

November 28, 2019

1 Fully-Connected Neural Nets

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

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

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

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

  return out, cache
```

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

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

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

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

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

[3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

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

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

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

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

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

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

2 Affine layer: foward

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

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

[5]:
```
# 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:  9.769847728806635e-10
```

3 Affine layer: backward

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

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

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

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

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

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

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

4 ReLU activation: forward

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

[7]:
```
# 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:

[8]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)
```

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

[FILL THIS IN]

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 `cs231n/layer_utils.py`.

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

[9]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

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

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w,␣
 ↪b)[0], x, 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:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs231n/layers.py`.

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

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

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

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

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

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

```
Testing svm_loss:
loss:  8.999602749096233
```

```
dx error:  1.4021566006651672e-09

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

8 Two-layer network

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

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

[11]:
```python
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

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

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

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

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

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

```
Testing initialization …
Testing test-time forward pass …
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 8.18e-07
W2 relative error: 7.98e-08
b1 relative error: 1.09e-09
b2 relative error: 7.76e-10
```

9 Solver

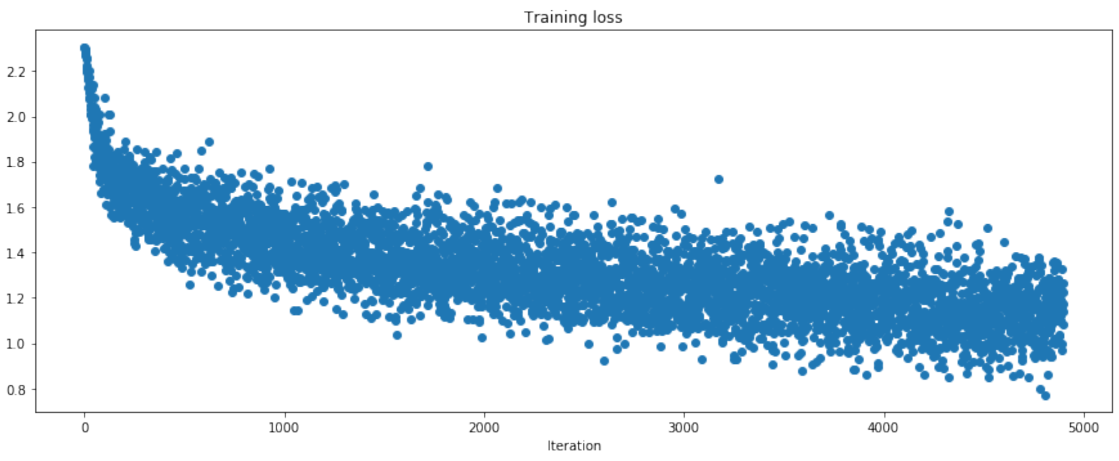
In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

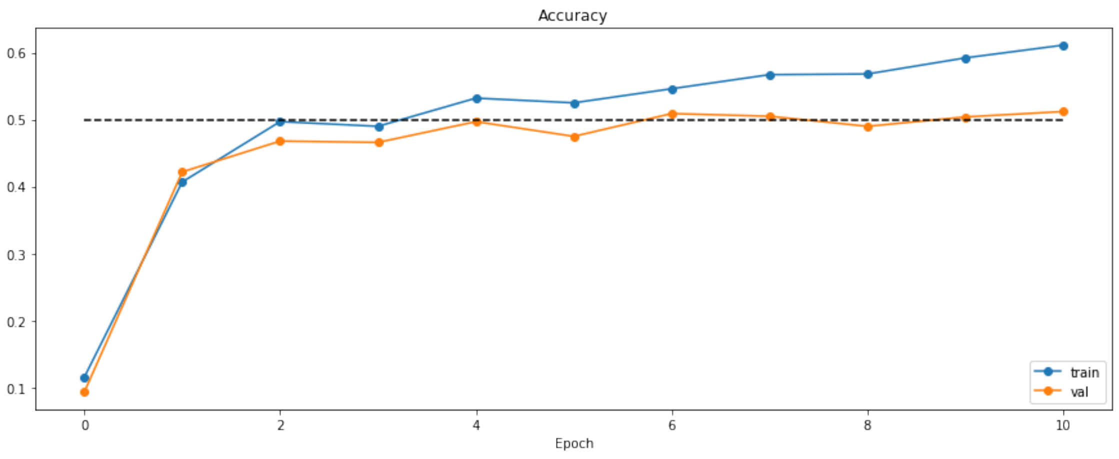
Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

[12]:
```
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
print('Solver Initiated...')
solver = Solver(model, data, lr_decay=0.95, optim_config={'learning_rate':
 ↪1e-3}, verbose=False)
solver.train()
print('Training Compeleted')
print('Best Validated Accuracy:', solver.best_val_acc)

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

```
Solver Initiated…
Training Compeleted
Best Validated Accuracy: 0.512
```

[13]:
```
# Run this cell to visualize training loss and train / val accuracy

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

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





10 Multilayer network

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

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

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

10.1 Initial loss and gradient check

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

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

[14]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))
```

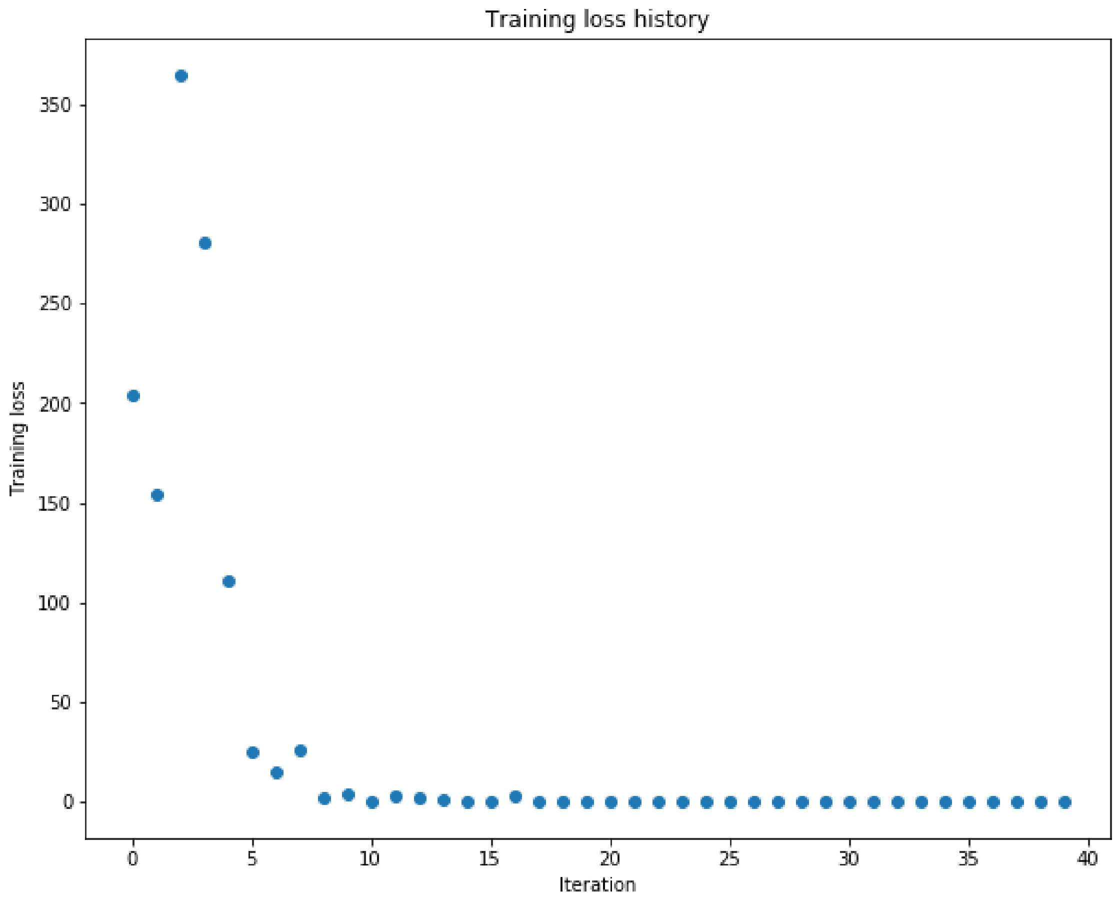
```
for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

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

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,␣
↪h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 6.86e-09
W2 relative error: 3.52e-08
W3 relative error: 1.32e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

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

[36]:
```
# 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 = 1e-1   # Experiment with this!
learning_rate = 1e-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, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 204.410346
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.135000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.109000
(Epoch 2 / 20) train acc: 0.380000; val_acc: 0.105000
(Epoch 3 / 20) train acc: 0.740000; val_acc: 0.136000
(Epoch 4 / 20) train acc: 0.880000; val_acc: 0.129000
(Epoch 5 / 20) train acc: 0.920000; val_acc: 0.123000
(Iteration 11 / 40) loss: 0.000000
(Epoch 6 / 20) train acc: 0.920000; val_acc: 0.127000
(Epoch 7 / 20) train acc: 0.980000; val_acc: 0.140000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.140000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.134000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.134000
(Iteration 21 / 40) loss: 0.000009
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.134000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.134000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.141000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.135000
(Iteration 31 / 40) loss: 0.000000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.135000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.135000
```

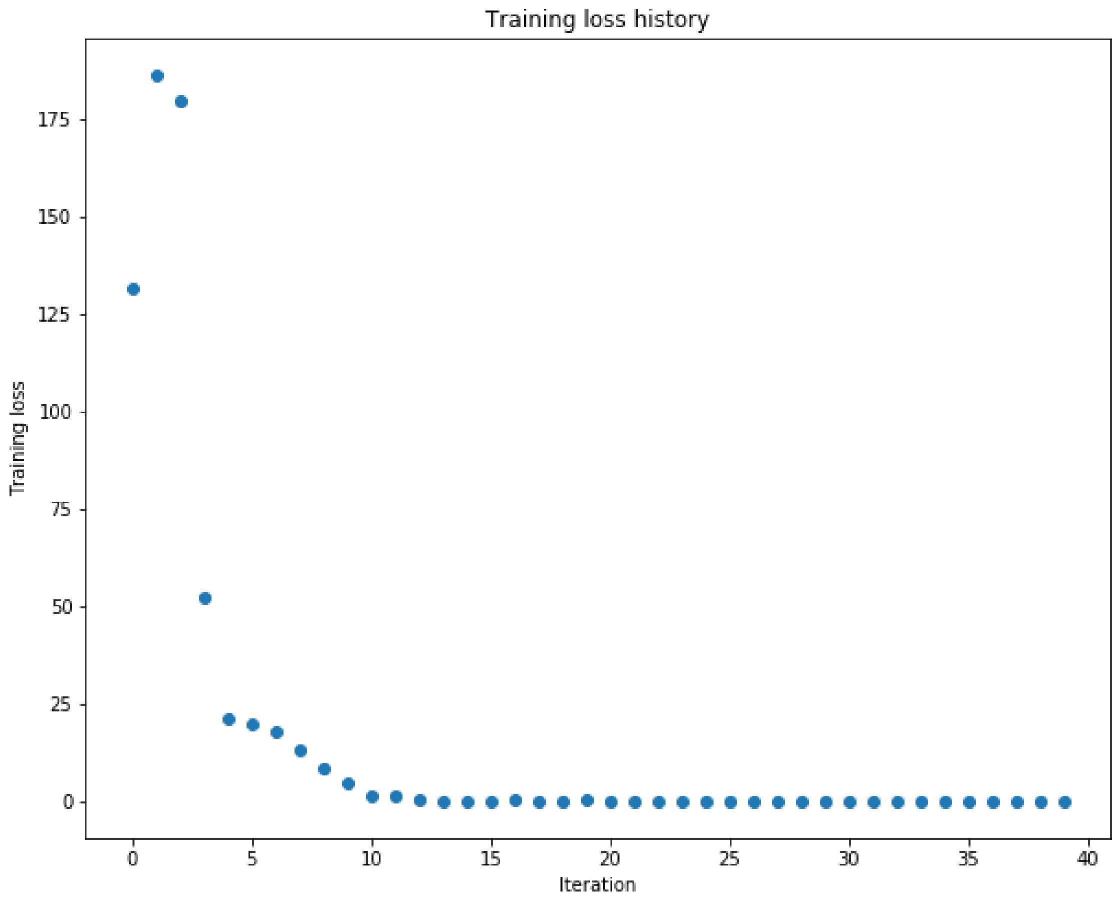


Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[38]:
```python
# 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 = 2e-3  # Experiment with this!
weight_scale = 1e-1   # 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, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 131.656266
(Epoch 0 / 20) train acc: 0.240000; val_acc: 0.096000
(Epoch 1 / 20) train acc: 0.160000; val_acc: 0.102000
(Epoch 2 / 20) train acc: 0.300000; val_acc: 0.096000
(Epoch 3 / 20) train acc: 0.360000; val_acc: 0.105000
(Epoch 4 / 20) train acc: 0.560000; val_acc: 0.135000
(Epoch 5 / 20) train acc: 0.800000; val_acc: 0.122000
(Iteration 11 / 40) loss: 1.254344
(Epoch 6 / 20) train acc: 0.920000; val_acc: 0.123000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.122000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.122000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.122000
(Iteration 21 / 40) loss: 0.002116
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.122000
(Iteration 31 / 40) loss: 0.000865
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.123000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.122000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.122000
```



10.2 Inline Question 2:

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

10.3 Answer:

[FILL THIS IN]

11 Update rules

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

12 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/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.

[18]:
```
from cs231n.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.

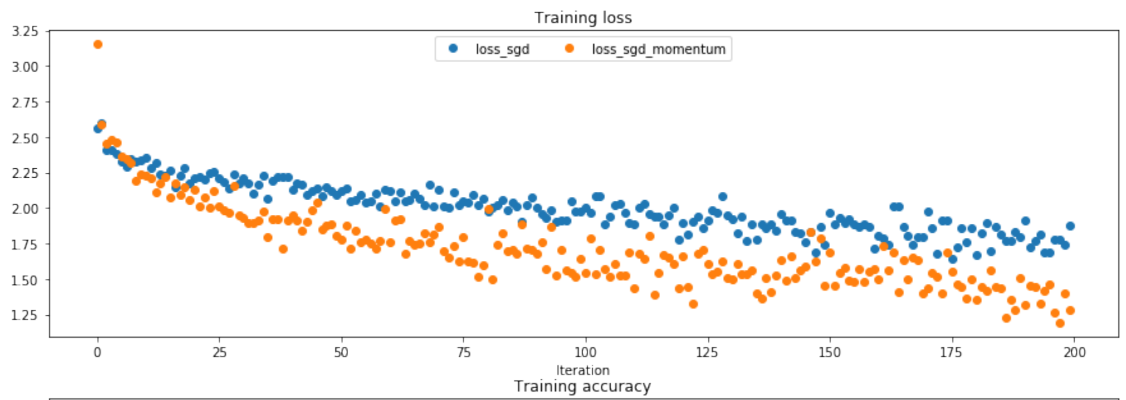
[19]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}
```

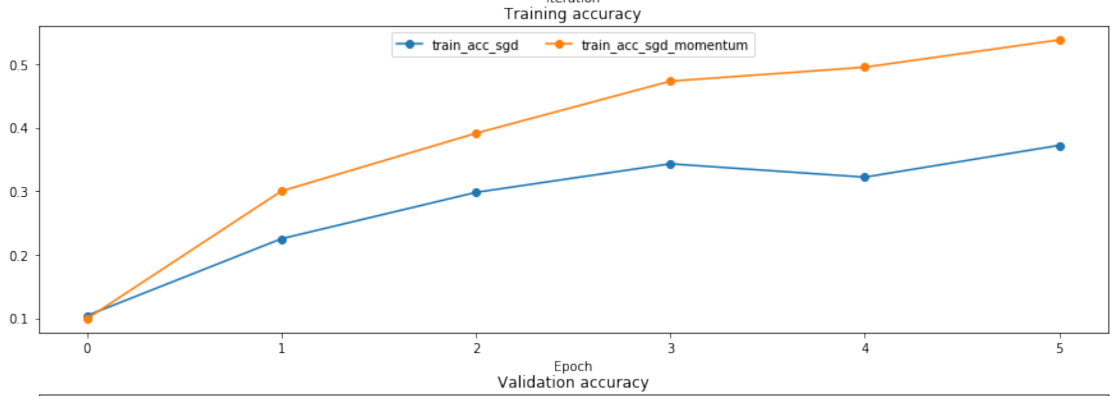
```
solvers = {}

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()
  print()

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

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

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

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

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

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

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

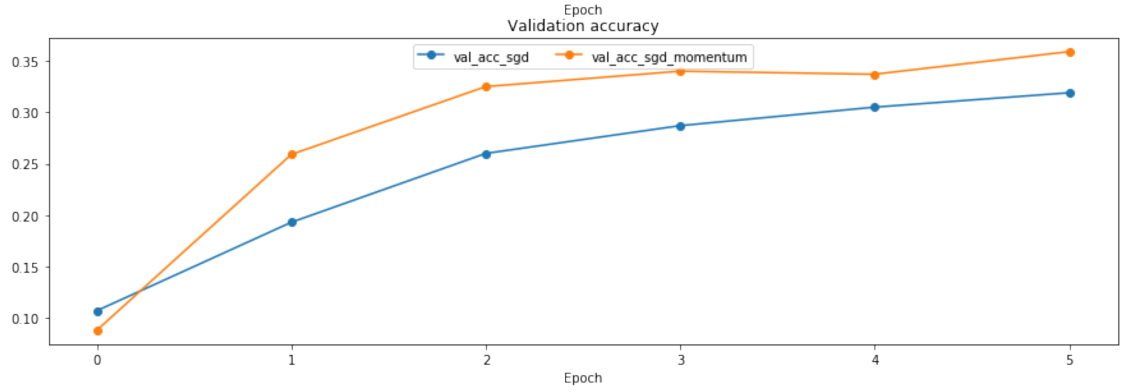
```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
```

```
(Epoch 0 / 5) train acc: 0.104000; val_acc: 0.107000
(Iteration 11 / 200) loss: 2.356069
(Iteration 21 / 200) loss: 2.214091
(Iteration 31 / 200) loss: 2.205928
(Epoch 1 / 5) train acc: 0.225000; val_acc: 0.193000
(Iteration 41 / 200) loss: 2.132095
(Iteration 51 / 200) loss: 2.118950
(Iteration 61 / 200) loss: 2.116443
(Iteration 71 / 200) loss: 2.132549
(Epoch 2 / 5) train acc: 0.298000; val_acc: 0.260000
(Iteration 81 / 200) loss: 1.977227
(Iteration 91 / 200) loss: 2.007528
(Iteration 101 / 200) loss: 2.004762
(Iteration 111 / 200) loss: 1.885342
(Epoch 3 / 5) train acc: 0.343000; val_acc: 0.287000
(Iteration 121 / 200) loss: 1.891516
(Iteration 131 / 200) loss: 1.923677
(Iteration 141 / 200) loss: 1.957744
(Iteration 151 / 200) loss: 1.966736
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.305000
(Iteration 161 / 200) loss: 1.801483
(Iteration 171 / 200) loss: 1.973779
(Iteration 181 / 200) loss: 1.666572
(Iteration 191 / 200) loss: 1.909494
(Epoch 5 / 5) train acc: 0.372000; val_acc: 0.319000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153778
(Epoch 0 / 5) train acc: 0.099000; val_acc: 0.088000
(Iteration 11 / 200) loss: 2.227203
(Iteration 21 / 200) loss: 2.125322
(Iteration 31 / 200) loss: 1.933623
(Epoch 1 / 5) train acc: 0.300000; val_acc: 0.259000
(Iteration 41 / 200) loss: 1.951480
(Iteration 51 / 200) loss: 1.778344
(Iteration 61 / 200) loss: 1.759060
(Iteration 71 / 200) loss: 1.865580
(Epoch 2 / 5) train acc: 0.391000; val_acc: 0.325000
(Iteration 81 / 200) loss: 1.997256
(Iteration 91 / 200) loss: 1.675952
(Iteration 101 / 200) loss: 1.539517
(Iteration 111 / 200) loss: 1.437328
(Epoch 3 / 5) train acc: 0.473000; val_acc: 0.340000
(Iteration 121 / 200) loss: 1.660326
(Iteration 131 / 200) loss: 1.495063
(Iteration 141 / 200) loss: 1.632314
(Iteration 151 / 200) loss: 1.686809
(Epoch 4 / 5) train acc: 0.495000; val_acc: 0.337000
```

```
(Iteration 161 / 200) loss: 1.495090
(Iteration 171 / 200) loss: 1.432555
(Iteration 181 / 200) loss: 1.352575
(Iteration 191 / 200) loss: 1.314671
(Epoch 5 / 5) train acc: 0.538000; val_acc: 0.359000


D:\Anaconda\lib\site-packages\ipykernel_launcher.py:39:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:42:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:45:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:49:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```







13 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 `cs231n/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.

[20]:
```
# Test RMSProp implementation
from cs231n.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
```

[21]:
```
# Test Adam implementation
from cs231n.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:

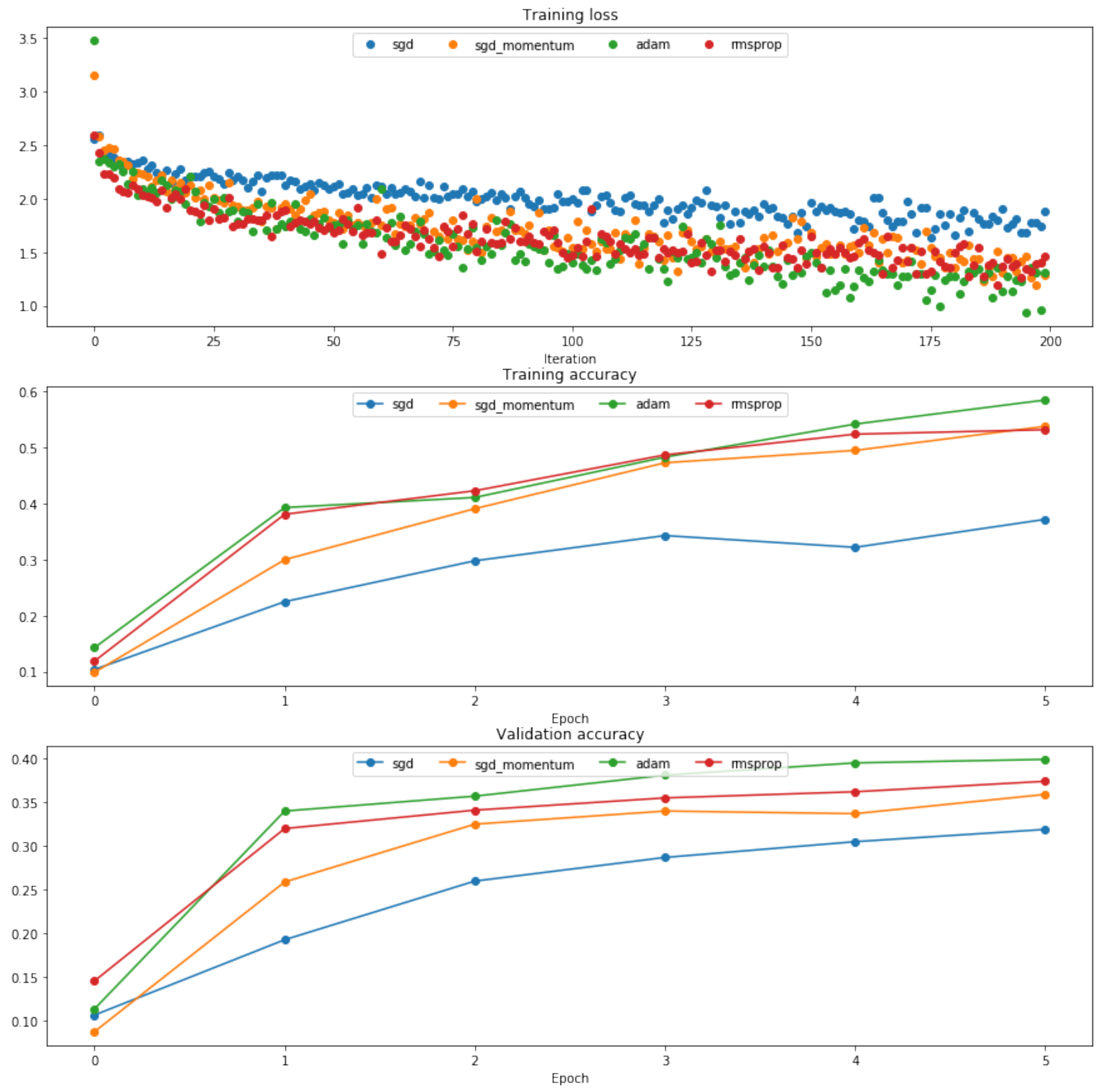
[22]:
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  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()

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

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

plt.subplot(3, 1, 3)
```

```python
plt.title('Validation accuracy')
plt.xlabel('Epoch')

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

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

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

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

```
running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.143000; val_acc: 0.114000
(Iteration 11 / 200) loss: 2.089203
(Iteration 21 / 200) loss: 2.211850
(Iteration 31 / 200) loss: 1.786014
(Epoch 1 / 5) train acc: 0.393000; val_acc: 0.340000
(Iteration 41 / 200) loss: 1.743813
(Iteration 51 / 200) loss: 1.752165
(Iteration 61 / 200) loss: 2.095686
(Iteration 71 / 200) loss: 1.489003
(Epoch 2 / 5) train acc: 0.411000; val_acc: 0.357000
(Iteration 81 / 200) loss: 1.546641
(Iteration 91 / 200) loss: 1.412223
(Iteration 101 / 200) loss: 1.401821
(Iteration 111 / 200) loss: 1.518779
(Epoch 3 / 5) train acc: 0.483000; val_acc: 0.381000
(Iteration 121 / 200) loss: 1.234374
(Iteration 131 / 200) loss: 1.452269
(Iteration 141 / 200) loss: 1.366984
(Iteration 151 / 200) loss: 1.500379
(Epoch 4 / 5) train acc: 0.542000; val_acc: 0.395000
(Iteration 161 / 200) loss: 1.335077
(Iteration 171 / 200) loss: 1.278162
(Iteration 181 / 200) loss: 1.271657
(Iteration 191 / 200) loss: 1.135909
(Epoch 5 / 5) train acc: 0.585000; val_acc: 0.399000
```

```
running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895732
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.486923
(Iteration 71 / 200) loss: 1.628511
(Epoch 2 / 5) train acc: 0.423000; val_acc: 0.341000
(Iteration 81 / 200) loss: 1.506182
(Iteration 91 / 200) loss: 1.600674
(Iteration 101 / 200) loss: 1.478501
(Iteration 111 / 200) loss: 1.577709
(Epoch 3 / 5) train acc: 0.487000; val_acc: 0.355000
(Iteration 121 / 200) loss: 1.495931
(Iteration 131 / 200) loss: 1.525799
(Iteration 141 / 200) loss: 1.552580
(Iteration 151 / 200) loss: 1.654283
(Epoch 4 / 5) train acc: 0.524000; val_acc: 0.362000
(Iteration 161 / 200) loss: 1.589371
(Iteration 171 / 200) loss: 1.413529
(Iteration 181 / 200) loss: 1.500273
(Iteration 191 / 200) loss: 1.365943
(Epoch 5 / 5) train acc: 0.532000; val_acc: 0.374000


D:\Anaconda\lib\site-packages\ipykernel_launcher.py:30:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:33:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:36:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
D:\Anaconda\lib\site-packages\ipykernel_launcher.py:40:
```

```
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```



13.1 Inline Question 3:

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?

13.2 Answer:

[FILL THIS IN]

14 Train a good model!

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

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

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

[25]:
```
best_model = None
best_acc = -1
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

num_train = 20000
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_rates = np.logspace(-3.8, -2.8, 8)
weight_scales = np.logspace(-2, -1.6, 4)



for lr in learning_rates:
    for ws in weight_scales:

        model = FullyConnectedNet([100, 85, 80, 80], weight_scale=ws,
                                  dtype=np.float64)

        solver = Solver(model, small_data,
                        num_epochs=2, batch_size=200,
                        update_rule='adam',
                        optim_config={
                        'learning_rate': lr,
                        },
                        verbose=False)
        solver.train()
        if solver.best_val_acc > best_acc:
            best_model = model
            best_acc = solver.best_val_acc
            best_lr, best_ws = (lr, ws)
            print("best valid acc yet: %f   lr: %e ws: %e" % (best_acc, lr, ws))


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

```
best valid acc yet: 0.352000   lr: 1.584893e-04 ws: 1.000000e-02
best valid acc yet: 0.395000   lr: 1.584893e-04 ws: 1.359356e-02
best valid acc yet: 0.412000   lr: 1.584893e-04 ws: 2.511886e-02
best valid acc yet: 0.424000   lr: 2.202202e-04 ws: 1.847850e-02
best valid acc yet: 0.434000   lr: 2.202202e-04 ws: 2.511886e-02
best valid acc yet: 0.435000   lr: 4.251786e-04 ws: 1.847850e-02
best valid acc yet: 0.448000   lr: 4.251786e-04 ws: 2.511886e-02
best valid acc yet: 0.469000   lr: 8.208914e-04 ws: 2.511886e-02
```

15 Test your model!

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

[26]:
```
best_lr = 8.208914e-04
best_ws = 1.847850e-02

best_model = FullyConnectedNet([85, 85, 80, 80], weight_scale=best_ws,
                                  dtype=np.float64)

solver = Solver(best_model, data,
                num_epochs=5, batch_size=200,
                update_rule='adam',
                optim_config={
                'learning_rate': best_lr,
                },
                verbose=True)
solver.train()
```

```python
print("BEST VALID ACC: %f" % solver.best_val_acc)
```

```
(Iteration 1 / 1225) loss: 2.301771
(Epoch 0 / 5) train acc: 0.136000; val_acc: 0.160000
(Iteration 11 / 1225) loss: 2.038054
(Iteration 21 / 1225) loss: 2.006881
(Iteration 31 / 1225) loss: 1.869662
(Iteration 41 / 1225) loss: 1.799201
(Iteration 51 / 1225) loss: 1.803220
(Iteration 61 / 1225) loss: 1.843045
(Iteration 71 / 1225) loss: 1.802073
(Iteration 81 / 1225) loss: 1.693561
(Iteration 91 / 1225) loss: 1.673529
(Iteration 101 / 1225) loss: 1.811663
(Iteration 111 / 1225) loss: 1.651202
(Iteration 121 / 1225) loss: 1.658457
(Iteration 131 / 1225) loss: 1.708132
(Iteration 141 / 1225) loss: 1.589373
(Iteration 151 / 1225) loss: 1.659702
(Iteration 161 / 1225) loss: 1.745562
(Iteration 171 / 1225) loss: 1.540619
(Iteration 181 / 1225) loss: 1.545648
(Iteration 191 / 1225) loss: 1.628894
(Iteration 201 / 1225) loss: 1.627824
(Iteration 211 / 1225) loss: 1.557569
(Iteration 221 / 1225) loss: 1.481052
(Iteration 231 / 1225) loss: 1.678927
(Iteration 241 / 1225) loss: 1.537228
(Epoch 1 / 5) train acc: 0.493000; val_acc: 0.464000
(Iteration 251 / 1225) loss: 1.499582
(Iteration 261 / 1225) loss: 1.450262
(Iteration 271 / 1225) loss: 1.377008
(Iteration 281 / 1225) loss: 1.582571
(Iteration 291 / 1225) loss: 1.459169
(Iteration 301 / 1225) loss: 1.411300
(Iteration 311 / 1225) loss: 1.539632
(Iteration 321 / 1225) loss: 1.733497
(Iteration 331 / 1225) loss: 1.369877
(Iteration 341 / 1225) loss: 1.571904
(Iteration 351 / 1225) loss: 1.569042
(Iteration 361 / 1225) loss: 1.526635
(Iteration 371 / 1225) loss: 1.470219
(Iteration 381 / 1225) loss: 1.594407
(Iteration 391 / 1225) loss: 1.430872
(Iteration 401 / 1225) loss: 1.314062
(Iteration 411 / 1225) loss: 1.608600
(Iteration 421 / 1225) loss: 1.490264
```

```
(Iteration 431 / 1225) loss: 1.454586
(Iteration 441 / 1225) loss: 1.471431
(Iteration 451 / 1225) loss: 1.380531
(Iteration 461 / 1225) loss: 1.489990
(Iteration 471 / 1225) loss: 1.333537
(Iteration 481 / 1225) loss: 1.389622
(Epoch 2 / 5) train acc: 0.517000; val_acc: 0.482000
(Iteration 491 / 1225) loss: 1.532352
(Iteration 501 / 1225) loss: 1.386140
(Iteration 511 / 1225) loss: 1.393768
(Iteration 521 / 1225) loss: 1.370802
(Iteration 531 / 1225) loss: 1.459193
(Iteration 541 / 1225) loss: 1.546071
(Iteration 551 / 1225) loss: 1.528938
(Iteration 561 / 1225) loss: 1.412577
(Iteration 571 / 1225) loss: 1.291795
(Iteration 581 / 1225) loss: 1.343924
(Iteration 591 / 1225) loss: 1.392638
(Iteration 601 / 1225) loss: 1.521811
(Iteration 611 / 1225) loss: 1.449436
(Iteration 621 / 1225) loss: 1.359829
(Iteration 631 / 1225) loss: 1.314342
(Iteration 641 / 1225) loss: 1.413525
(Iteration 651 / 1225) loss: 1.319351
(Iteration 661 / 1225) loss: 1.467216
(Iteration 671 / 1225) loss: 1.386705
(Iteration 681 / 1225) loss: 1.493161
(Iteration 691 / 1225) loss: 1.363820
(Iteration 701 / 1225) loss: 1.351871
(Iteration 711 / 1225) loss: 1.269137
(Iteration 721 / 1225) loss: 1.317871
(Iteration 731 / 1225) loss: 1.160985
(Epoch 3 / 5) train acc: 0.524000; val_acc: 0.477000
(Iteration 741 / 1225) loss: 1.330009
(Iteration 751 / 1225) loss: 1.399085
(Iteration 761 / 1225) loss: 1.184626
(Iteration 771 / 1225) loss: 1.294741
(Iteration 781 / 1225) loss: 1.291185
(Iteration 791 / 1225) loss: 1.455089
(Iteration 801 / 1225) loss: 1.250871
(Iteration 811 / 1225) loss: 1.328729
(Iteration 821 / 1225) loss: 1.235982
(Iteration 831 / 1225) loss: 1.410605
(Iteration 841 / 1225) loss: 1.344086
(Iteration 851 / 1225) loss: 1.435071
(Iteration 861 / 1225) loss: 1.479619
(Iteration 871 / 1225) loss: 1.176542
(Iteration 881 / 1225) loss: 1.260821
```

```
(Iteration 891 / 1225) loss: 1.416800
(Iteration 901 / 1225) loss: 1.327771
(Iteration 911 / 1225) loss: 1.268409
(Iteration 921 / 1225) loss: 1.355131
(Iteration 931 / 1225) loss: 1.147941
(Iteration 941 / 1225) loss: 1.412509
(Iteration 951 / 1225) loss: 1.287355
(Iteration 961 / 1225) loss: 1.262990
(Iteration 971 / 1225) loss: 1.288821
(Epoch 4 / 5) train acc: 0.536000; val_acc: 0.509000
(Iteration 981 / 1225) loss: 1.252663
(Iteration 991 / 1225) loss: 1.234500
(Iteration 1001 / 1225) loss: 1.298653
(Iteration 1011 / 1225) loss: 1.444061
(Iteration 1021 / 1225) loss: 1.233633
(Iteration 1031 / 1225) loss: 1.392105
(Iteration 1041 / 1225) loss: 1.319189
(Iteration 1051 / 1225) loss: 1.247098
(Iteration 1061 / 1225) loss: 1.411017
(Iteration 1071 / 1225) loss: 1.330203
(Iteration 1081 / 1225) loss: 1.269069
(Iteration 1091 / 1225) loss: 1.176891
(Iteration 1101 / 1225) loss: 1.214366
(Iteration 1111 / 1225) loss: 1.162532
(Iteration 1121 / 1225) loss: 1.330445
(Iteration 1131 / 1225) loss: 1.214539
(Iteration 1141 / 1225) loss: 1.396501
(Iteration 1151 / 1225) loss: 1.236271
(Iteration 1161 / 1225) loss: 1.169584
(Iteration 1171 / 1225) loss: 1.254802
(Iteration 1181 / 1225) loss: 1.305022
(Iteration 1191 / 1225) loss: 1.163972
(Iteration 1201 / 1225) loss: 1.278898
(Iteration 1211 / 1225) loss: 1.254015
(Iteration 1221 / 1225) loss: 1.150603
(Epoch 5 / 5) train acc: 0.554000; val_acc: 0.504000
BEST VALID ACC: 0.509000
```

[27]:
```python
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:  0.509
Test set accuracy:  0.48
```