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 gr
adients

  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 outp
uts) 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.

In [22]:

```
# 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, eva
l_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default siz
e 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 rel
oad it, use:
  %reload_ext autoreload
```

In [23]:

```
# Load the (preprocessed) CIFAR10 data.

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

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

Affine layer: 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:

In [24]:

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

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

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

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

Affine layer: backward

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

In [25]:

```
# 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_forwar
d(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forwar
d(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forwar
d(x, w, b)[0], b, dout)

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

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

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

ReLU activation: forward

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

In [26]:

```
# 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 or
der of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

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

ReLU activation: backward

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

In [27]:

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

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

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

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

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

Inline Question 1:

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

1. Sigmoid
2. ReLU
3. Leaky ReLU

Answer:

1. Sigmoid function suffers from the vanishing gradient problem because the gradient takes small values in the tails of the function (close to 0), this can easily be observed by looking at limits of the derivative of the sigmoid function. A one dimensional example is to saturate the function, e.g., [-1e5,1e5] ReLU's gradient is 0 or 1. Thus it can suffer from the vanishing gradient problem when all input values are negative. However, having all negative inputs is less probable. Because of this problem occurs only in one side (negative values), it is also known as the "dying ReLU problem", this name comes from the fact that some neurons will never be updated, i.e., they will be "dead". A one dimensional example that can vanish the gradient is to consider only negative values, e.g., [-1,-2,-4,-5]. Leaky ReLU tries to solve the ReLU problem of "dead" neurons by considering small negative slope when we have negative values, i.e., if $x < 0$ then alpha else x, where alpha can be equal to 0.01. A one dimensional example that can vanish the gradient is to consider all zero values, e.g., [0,0,0,0], it can only happen with a bad network initialization.

"Sandwich" layers

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

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

In [10]:

```
from cs231n.layer_utils import affine_relu_forward, affine_rel
u_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_f
orward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_f
orward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_f
orward(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_backwa
rd:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

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:

In [11]:

```
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
```

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.

In [12]:

```
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 rig
ht'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem ri
ght'
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)
```

```
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 pas
s'

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 trainin
g-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 regular
ization 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_nu
m, grads[name])))
```

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

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.

In [14]:

```
model = TwoLayerNet()
solver = None

################################################################################
################
# TODO: Use a Solver instance to train a TwoLayerNet that achi
eves at least  #
# 50% accuracy on the validation set.
#
################################################################################
################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****
*

model = TwoLayerNet(input_dim=3*32*32, hidden_dim=100, num_cla
sses=10,
                    weight_scale=1e-3, reg=0.7)

solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.9,
                num_epochs=10, batch_size=100,
                print_every=100)
solver.train()

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

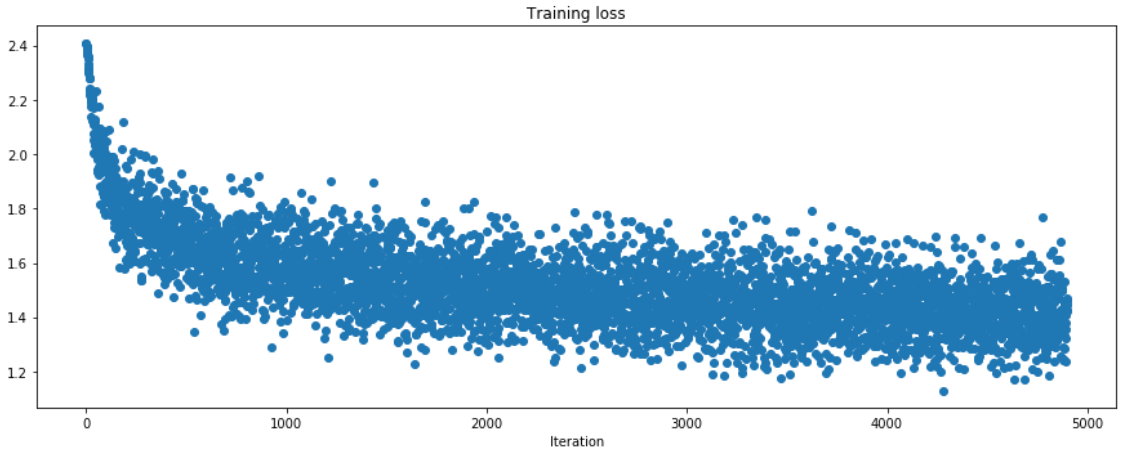
```
(Iteration 1 / 4900) loss: 2.408314
(Epoch 0 / 10) train acc: 0.141000; val_acc: 0.154
000
(Iteration 101 / 4900) loss: 1.945939
(Iteration 201 / 4900) loss: 1.780133
(Iteration 301 / 4900) loss: 1.806620
(Iteration 401 / 4900) loss: 1.807101
(Epoch 1 / 10) train acc: 0.445000; val_acc: 0.430
000
(Iteration 501 / 4900) loss: 1.544380
```

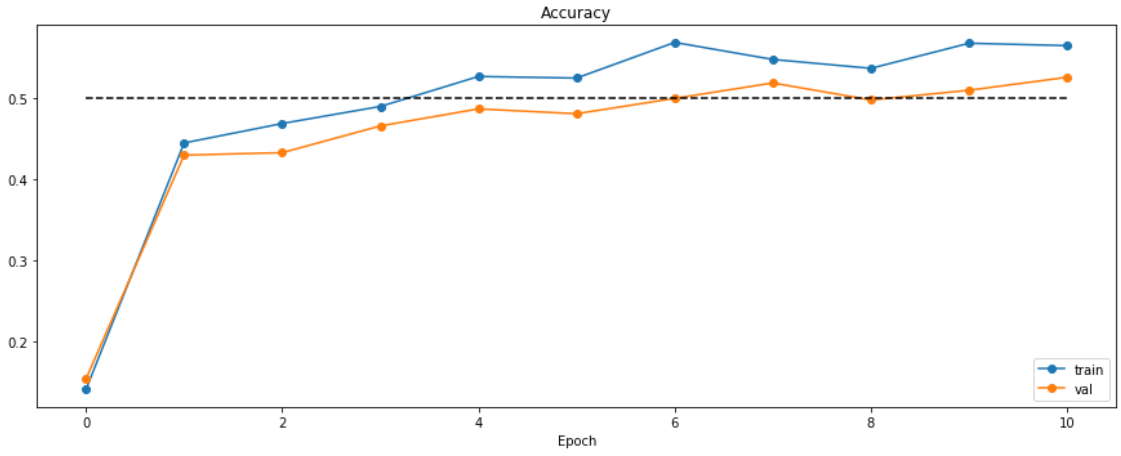
```
(Iteration 601 / 4900) loss: 1.769243
(Iteration 701 / 4900) loss: 1.731019
(Iteration 801 / 4900) loss: 1.900799
(Iteration 901 / 4900) loss: 1.491739
(Epoch 2 / 10) train acc: 0.469000; val_acc: 0.433
000
(Iteration 1001 / 4900) loss: 1.561765
(Iteration 1101 / 4900) loss: 1.582483
(Iteration 1201 / 4900) loss: 1.568426
(Iteration 1301 / 4900) loss: 1.507251
(Iteration 1401 / 4900) loss: 1.726489
(Epoch 3 / 10) train acc: 0.490000; val_acc: 0.466
000
(Iteration 1501 / 4900) loss: 1.510903
(Iteration 1601 / 4900) loss: 1.491079
(Iteration 1701 / 4900) loss: 1.559660
(Iteration 1801 / 4900) loss: 1.357199
(Iteration 1901 / 4900) loss: 1.596737
(Epoch 4 / 10) train acc: 0.527000; val_acc: 0.487
000
(Iteration 2001 / 4900) loss: 1.467299
(Iteration 2101 / 4900) loss: 1.769517
(Iteration 2201 / 4900) loss: 1.442836
(Iteration 2301 / 4900) loss: 1.311903
(Iteration 2401 / 4900) loss: 1.427128
(Epoch 5 / 10) train acc: 0.525000; val_acc: 0.481
000
(Iteration 2501 / 4900) loss: 1.336258
(Iteration 2601 / 4900) loss: 1.452549
(Iteration 2701 / 4900) loss: 1.560751
(Iteration 2801 / 4900) loss: 1.593029
(Iteration 2901 / 4900) loss: 1.750660
(Epoch 6 / 10) train acc: 0.569000; val_acc: 0.500
000
(Iteration 3001 / 4900) loss: 1.608443
(Iteration 3101 / 4900) loss: 1.502184
(Iteration 3201 / 4900) loss: 1.231901
(Iteration 3301 / 4900) loss: 1.438971
(Iteration 3401 / 4900) loss: 1.526785
(Epoch 7 / 10) train acc: 0.548000; val_acc: 0.519
000
(Iteration 3501 / 4900) loss: 1.434022
(Iteration 3601 / 4900) loss: 1.402325
(Iteration 3701 / 4900) loss: 1.420445
(Iteration 3801 / 4900) loss: 1.483041
(Iteration 3901 / 4900) loss: 1.425493
(Epoch 8 / 10) train acc: 0.537000; val_acc: 0.498
000
```

```
(Iteration 4001 / 4900) loss: 1.523308
(Iteration 4101 / 4900) loss: 1.389707
(Iteration 4201 / 4900) loss: 1.363503
(Iteration 4301 / 4900) loss: 1.446814
(Iteration 4401 / 4900) loss: 1.328505
(Epoch 9 / 10) train acc: 0.568000; val_acc: 0.510
000
(Iteration 4501 / 4900) loss: 1.388716
(Iteration 4601 / 4900) loss: 1.522012
(Iteration 4701 / 4900) loss: 1.422739
(Iteration 4801 / 4900) loss: 1.288580
(Epoch 10 / 10) train acc: 0.565000; val_acc: 0.52
6000
```

In [19]:

```
# Run this cell to visualize training loss and train / val acc
uracy

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()
```





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.

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.

In [29]:

```
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 smalle
r.
  # NOTE: It is fine however to see an error for W2 on the ord
er 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_nu
m, grads[name])))
```

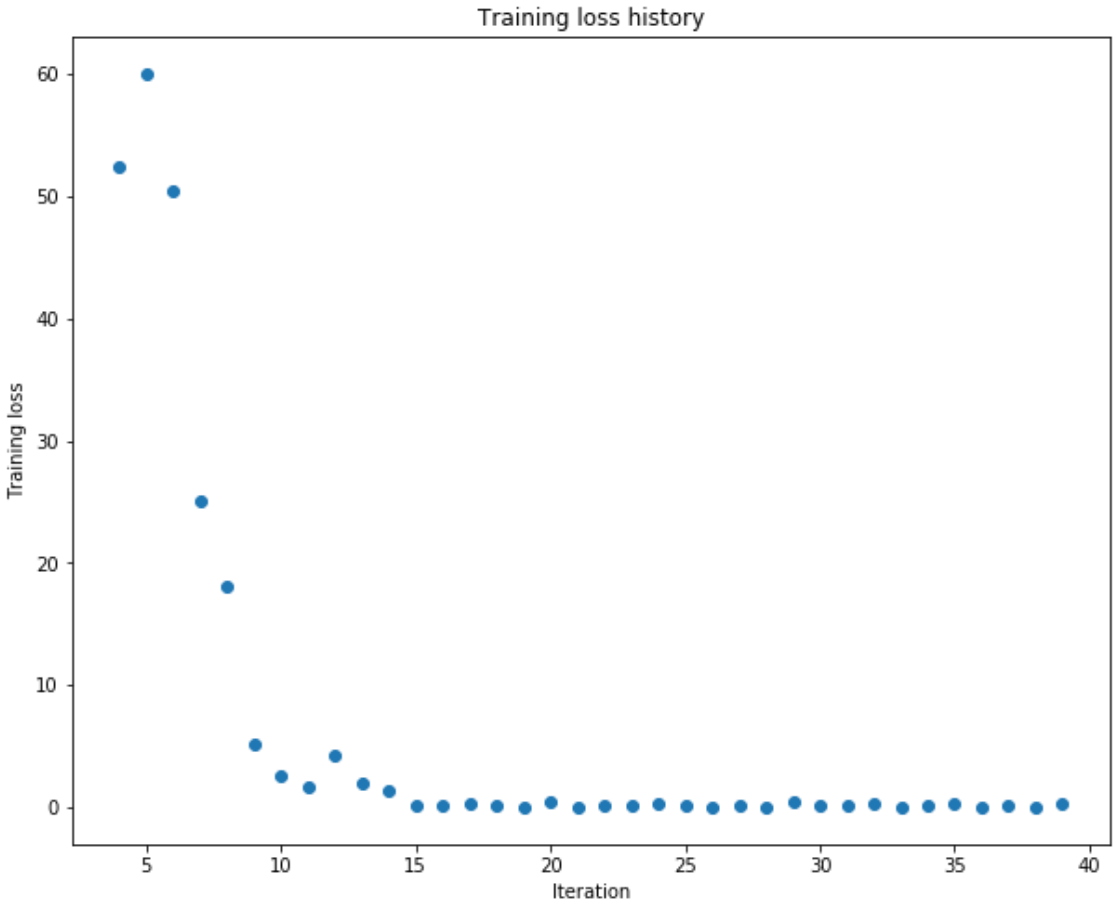
```
Running check with reg =  0
Initial loss:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 1.69e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 1.41e-08
W2 relative error: 6.87e-08
W3 relative error: 2.13e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 2.38e-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.

In [33]:

```
# 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 = 2e-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: inf
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.113
000
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.092
000
(Epoch 2 / 20) train acc: 0.420000; val_acc: 0.133
000
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.131
000
(Epoch 4 / 20) train acc: 0.620000; val_acc: 0.135
000
(Epoch 5 / 20) train acc: 0.800000; val_acc: 0.129
000
(Iteration 11 / 40) loss: 2.571154
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.130
000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.128
000
(Epoch 8 / 20) train acc: 0.920000; val_acc: 0.130
000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.123
000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.12
3000
(Iteration 21 / 40) loss: 0.433322
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Epoch 15 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Iteration 31 / 40) loss: 0.220346
(Epoch 16 / 20) train acc: 0.960000; val_acc: 0.12
4000
(Epoch 17 / 20) train acc: 0.960000; val_acc: 0.12
5000
(Epoch 18 / 20) train acc: 0.960000; val_acc: 0.12
5000
(Epoch 19 / 20) train acc: 0.960000; val_acc: 0.12
5000
(Epoch 20 / 20) train acc: 0.960000; val_acc: 0.12
5000
```

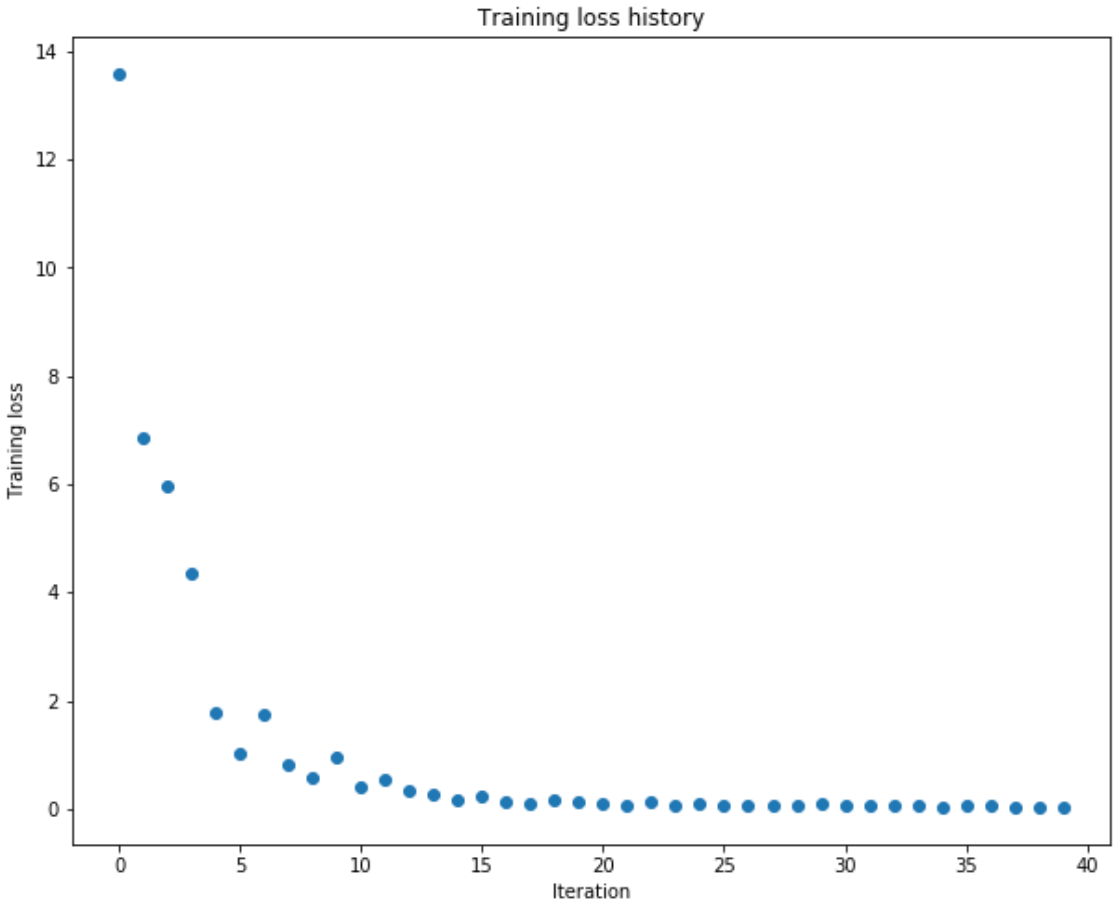


Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

In [36]:

```
# TODO: Use a five-layer Net to overfit 50 training examples b
y
# 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 = 5e-3  # Experiment with this!
weight_scale = 6e-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, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 13.584500
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.135
000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.115
000
(Epoch 2 / 20) train acc: 0.420000; val_acc: 0.115
000
(Epoch 3 / 20) train acc: 0.720000; val_acc: 0.116
000
(Epoch 4 / 20) train acc: 0.840000; val_acc: 0.114
000
(Epoch 5 / 20) train acc: 0.920000; val_acc: 0.117
000
(Iteration 11 / 40) loss: 0.406795
(Epoch 6 / 20) train acc: 0.960000; val_acc: 0.117
000
(Epoch 7 / 20) train acc: 0.980000; val_acc: 0.122
000
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.117
000
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.120
000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.11
7000
(Iteration 21 / 40) loss: 0.094295
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.11
5000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.11
9000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.12
2000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.12
2000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.12
4000
(Iteration 31 / 40) loss: 0.052302
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.12
6000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.12
1000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.11
9000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.11
9000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.11
1000
```



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?

Answer:

Comparing the difficulty of training these two networks, the five layer net was more sensitive to the initialization scale. In my opinion, it is due to the number of layers. As we have a deeper network, setting a small scale leads to vanishing the gradients (small products), on the other hand a large scale leads to exploding the gradients (large products), in deeper networks it occurs more frequently than in shallow networks. Finding a correct scale is harder than the three-layer net but can be obtained by using random search.

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.

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 (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.

In [37]:

```
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,  0.4077
5789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.7417
0526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.0756
5263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096
]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723
158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.6680
2105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.7388
1053],
  [ 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.

In [38]:

```
num_train = 4000
small_data = {
```

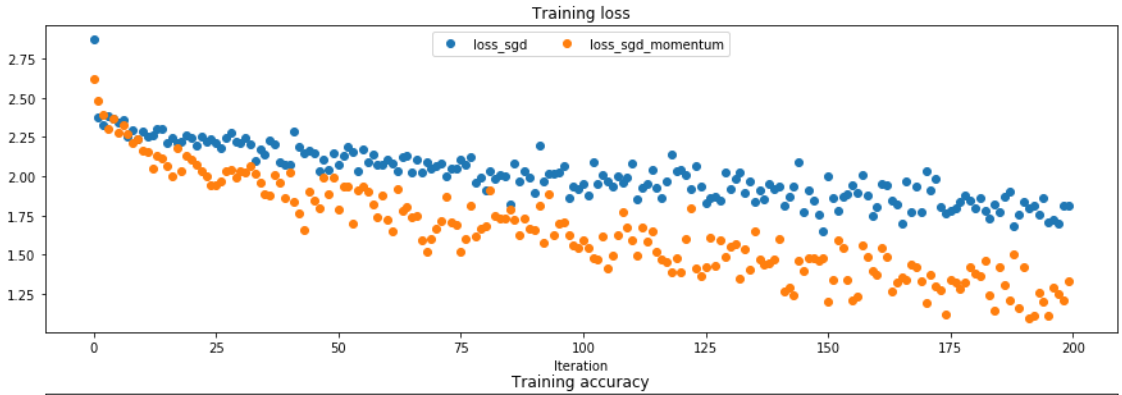
```
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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)
```

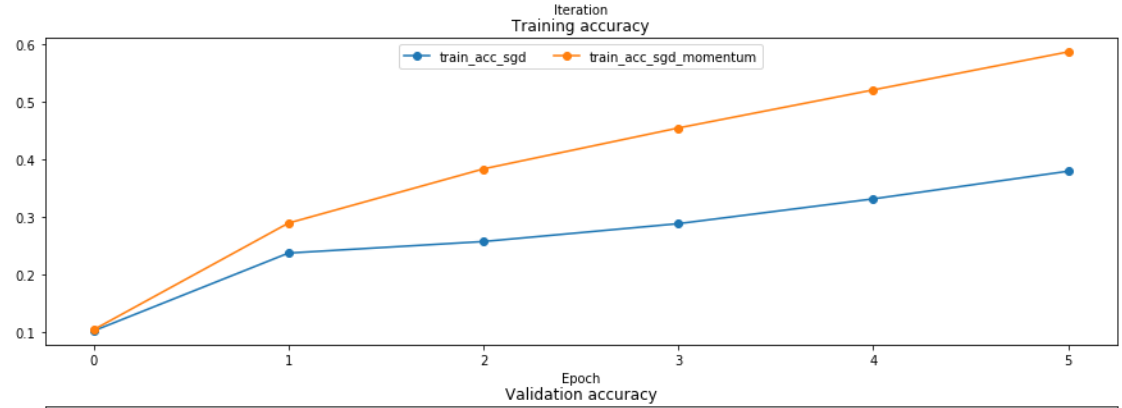
```
for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

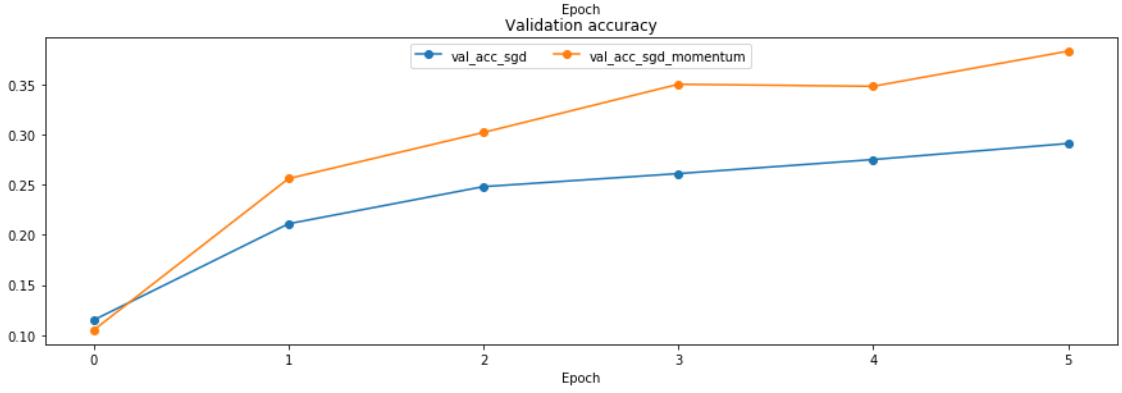
```
running with  sgd
(Iteration 1 / 200) loss: 2.870913
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.1150
00
(Iteration 11 / 200) loss: 2.286257
(Iteration 21 / 200) loss: 2.243553
(Iteration 31 / 200) loss: 2.215382
(Epoch 1 / 5) train acc: 0.238000; val_acc: 0.2110
00
(Iteration 41 / 200) loss: 2.073878
(Iteration 51 / 200) loss: 2.070358
(Iteration 61 / 200) loss: 2.108942
(Iteration 71 / 200) loss: 2.066009
(Epoch 2 / 5) train acc: 0.258000; val_acc: 0.2480
00
(Iteration 81 / 200) loss: 1.912479
(Iteration 91 / 200) loss: 1.892785
(Iteration 101 / 200) loss: 1.955628
(Iteration 111 / 200) loss: 2.081030
(Epoch 3 / 5) train acc: 0.289000; val_acc: 0.2610
00
(Iteration 121 / 200) loss: 2.043379
(Iteration 131 / 200) loss: 1.922446
(Iteration 141 / 200) loss: 1.938524
(Iteration 151 / 200) loss: 1.996964
(Epoch 4 / 5) train acc: 0.332000; val_acc: 0.2750
00
(Iteration 161 / 200) loss: 1.805984
(Iteration 171 / 200) loss: 2.032825
(Iteration 181 / 200) loss: 1.796822
(Iteration 191 / 200) loss: 1.840224
(Epoch 5 / 5) train acc: 0.380000; val_acc: 0.2910
00

running with  sgd_momentum
(Iteration 1 / 200) loss: 2.616879
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.1050
00
(Iteration 11 / 200) loss: 2.166215
(Iteration 21 / 200) loss: 2.108417
(Iteration 31 / 200) loss: 2.033840
```

```
(Epoch 1 / 5) train acc: 0.290000; val_acc: 0.2560
00
(Iteration 41 / 200) loss: 2.025788
(Iteration 51 / 200) loss: 1.791962
(Iteration 61 / 200) loss: 1.725599
(Iteration 71 / 200) loss: 1.663960
(Epoch 2 / 5) train acc: 0.384000; val_acc: 0.3020
00
(Iteration 81 / 200) loss: 1.682881
(Iteration 91 / 200) loss: 1.658188
(Iteration 101 / 200) loss: 1.596029
(Iteration 111 / 200) loss: 1.589211
(Epoch 3 / 5) train acc: 0.455000; val_acc: 0.3500
00
(Iteration 121 / 200) loss: 1.385709
(Iteration 131 / 200) loss: 1.548744
(Iteration 141 / 200) loss: 1.602561
(Iteration 151 / 200) loss: 1.204752
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.3480
00
(Iteration 161 / 200) loss: 1.374660
(Iteration 171 / 200) loss: 1.194586
(Iteration 181 / 200) loss: 1.382335
(Iteration 191 / 200) loss: 1.418928
(Epoch 5 / 5) train acc: 0.587000; val_acc: 0.3830
00
```

```
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:39: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:42: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:45: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:49: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
```







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.

In [39]:

```
# 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, -0.1846
7247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.0751
5774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.3353
2447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.5957
6619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.6580
4321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.7348
4377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.8130
2936],
  [ 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
```

In [40]:

```
# 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, -0.1906
0977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722
929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.3351
6969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.5980
1459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.6573
8853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.6046
7385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.5520
9767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.4996
6,    ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.5578
9474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.6552
6316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.7526
3158],
  [ 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:

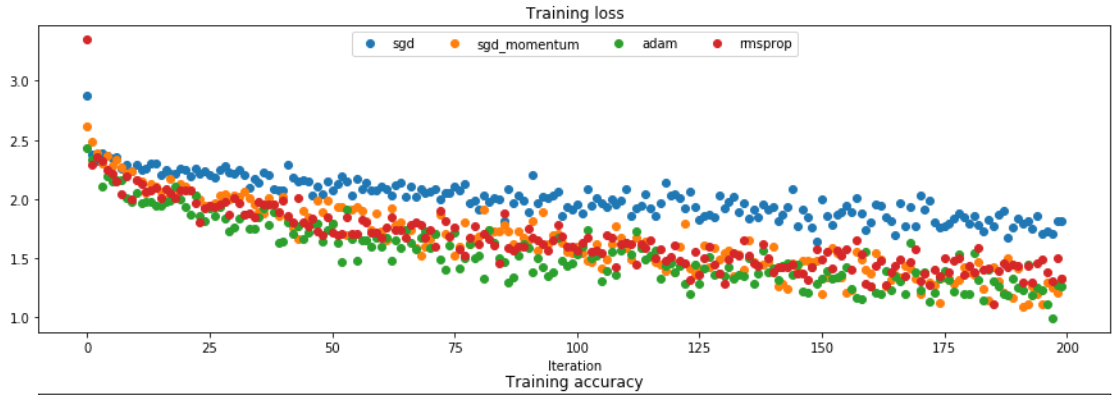
In [41]:

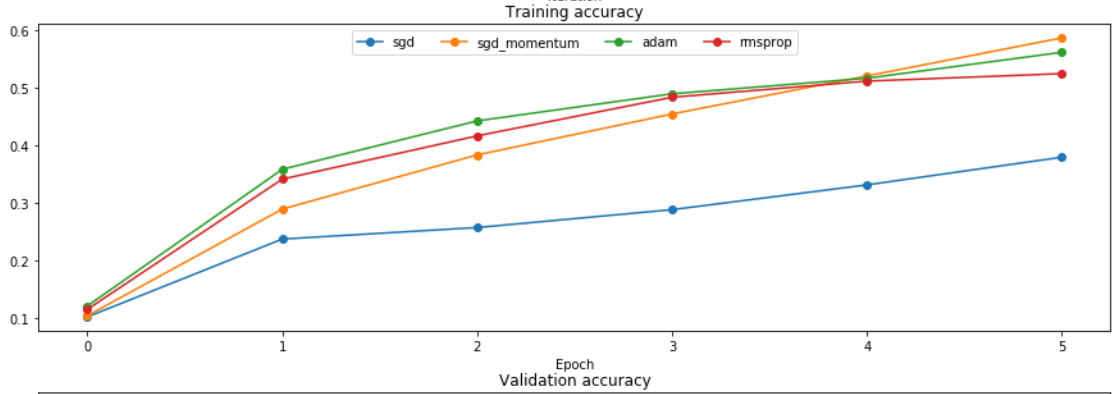
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_
scale=5e-2)

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

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

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

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

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

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

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

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

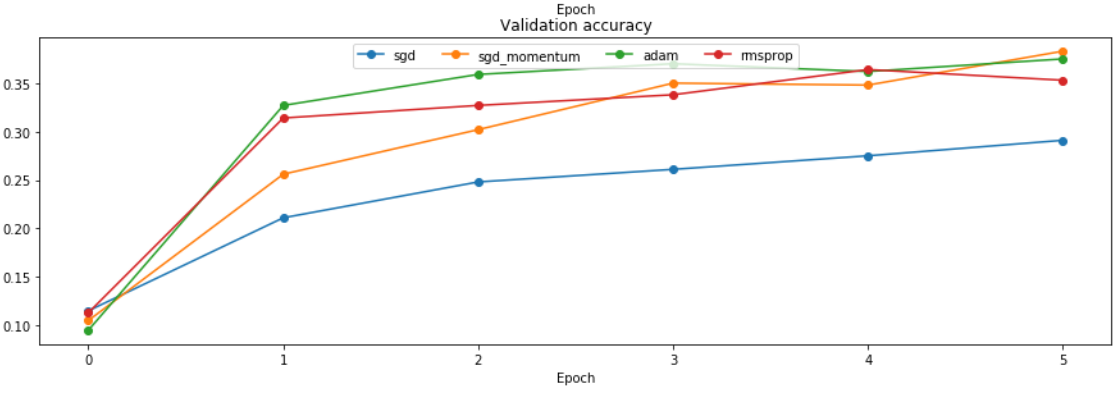
```
running with  adam
(Iteration 1 / 200) loss: 2.437399
(Epoch 0 / 5) train acc: 0.122000; val_acc: 0.0950
00
(Iteration 11 / 200) loss: 2.056220
(Iteration 21 / 200) loss: 1.936689
(Iteration 31 / 200) loss: 1.760988
(Epoch 1 / 5) train acc: 0.359000; val_acc: 0.3270
00
(Iteration 41 / 200) loss: 1.639867
(Iteration 51 / 200) loss: 1.796589
(Iteration 61 / 200) loss: 1.596225
(Iteration 71 / 200) loss: 1.591918
(Epoch 2 / 5) train acc: 0.443000; val_acc: 0.3590
00
(Iteration 81 / 200) loss: 1.529366
(Iteration 91 / 200) loss: 1.594191
(Iteration 101 / 200) loss: 1.535943
(Iteration 111 / 200) loss: 1.635587
(Epoch 3 / 5) train acc: 0.490000; val_acc: 0.3700
00
(Iteration 121 / 200) loss: 1.380302
(Iteration 131 / 200) loss: 1.353985
(Iteration 141 / 200) loss: 1.421162
(Iteration 151 / 200) loss: 1.326631
(Epoch 4 / 5) train acc: 0.517000; val_acc: 0.3620
00
(Iteration 161 / 200) loss: 1.267406
(Iteration 171 / 200) loss: 1.227904
(Iteration 181 / 200) loss: 1.204412
(Iteration 191 / 200) loss: 1.253392
(Epoch 5 / 5) train acc: 0.562000; val_acc: 0.3750
00

running with  rmsprop
(Iteration 1 / 200) loss: 3.347723
(Epoch 0 / 5) train acc: 0.116000; val_acc: 0.1130
00
(Iteration 11 / 200) loss: 2.163248
(Iteration 21 / 200) loss: 2.075757
(Iteration 31 / 200) loss: 2.008891
(Epoch 1 / 5) train acc: 0.342000; val_acc: 0.3140
00
(Iteration 41 / 200) loss: 1.986522
(Iteration 51 / 200) loss: 1.715517
(Iteration 61 / 200) loss: 1.680815
(Iteration 71 / 200) loss: 1.735764
(Epoch 2 / 5) train acc: 0.417000; val_acc: 0.3270
```

00
(Iteration 81 / 200) loss: 1.623309
(Iteration 91 / 200) loss: 1.751605
(Iteration 101 / 200) loss: 1.568823
(Iteration 111 / 200) loss: 1.631916
(Epoch 3 / 5) train acc: 0.484000; val_acc: 0.3380
00
(Iteration 121 / 200) loss: 1.607684
(Iteration 131 / 200) loss: 1.287405
(Iteration 141 / 200) loss: 1.378786
(Iteration 151 / 200) loss: 1.494665
(Epoch 4 / 5) train acc: 0.512000; val_acc: 0.3640
00
(Iteration 161 / 200) loss: 1.269931
(Iteration 171 / 200) loss: 1.304878
(Iteration 181 / 200) loss: 1.434893
(Iteration 191 / 200) loss: 1.438193
(Epoch 5 / 5) train acc: 0.525000; val_acc: 0.3530
00

```
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:30: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:33: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:36: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
/home/ec2-user/.conda/envs/cs231new/lib/python3.7/
site-packages/ipykernel_launcher.py:40: Matplotlib
DeprecationWarning: Adding an axes using the same
arguments as a previous axes currently reuses the
earlier instance.  In a future version, a new inst
ance will always be created and returned.  Meanwhi
le, this warning can be suppressed, and the future
behavior ensured, by passing a unique label to eac
h axes instance.
```


Training loss
sgd
sgd_momentum
adam
rmsprop
3.0
2.5
2.0
1.5
1.0
0
25
50
75
100
125
150
175
200
Iteration


Training accuracy
sgd
sgd_momentum
adam
rmsprop
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
3
4
5
Epoch


Validation accuracy
sgd
sgd_momentum
adam
rmsprop
0.35
0.30
0.25
0.20
0.15
0.10
0
1
2
3
4
5
Epoch

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?

Answer:

The update becomes very small because at each iteration we are adding up the squared gradients. Thus, if the number of iterations is large then the cache value will be very large because we are accumulating positive values (cache += dw**2). Therefore, when dividing the gradient (dw) by the square root of the cache value (np.sqrt(cache)) the update will be very small. Adam does not have this issue because it uses an exponential weighted average (EWA) of the squared gradients (this part is based on RMSprop). That is, it accumulates past gradients weighted exponentially, assigning large weights to recent gradients and small weights to old gradients. Besides that, Adam uses EWA of momentum instead of the default gradient (dw), resulting in large updates controlled by the EWA of squared gradients.

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.

In [43]:

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

model = FullyConnectedNet(
    [512, 256, 256, 256],
    weight_scale=1e-2,
    normalization='batchnorm',
    dropout=0.5
)
solver = Solver(
    model,
    data,
    num_epochs=10,
    print_every=100,
    batch_size=128,
    update_rule="adam",
    optim_config={'learning_rate': 1e-3},
    verbose=True
)
solver.train()
best_model = model

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

```
(Iteration 1 / 3820) loss: 2.317044
(Epoch 0 / 10) train acc: 0.111000; val_acc: 0.127
000
(Iteration 101 / 3820) loss: 1.897851
(Iteration 201 / 3820) loss: 1.652043
```

```
(Iteration 301 / 3820) loss: 1.614897
(Epoch 1 / 10) train acc: 0.421000; val_acc: 0.434
000
(Iteration 401 / 3820) loss: 1.676462
(Iteration 501 / 3820) loss: 1.681762
(Iteration 601 / 3820) loss: 1.743175
(Iteration 701 / 3820) loss: 1.539126
(Epoch 2 / 10) train acc: 0.463000; val_acc: 0.481
000
(Iteration 801 / 3820) loss: 1.621105
(Iteration 901 / 3820) loss: 1.659730
(Iteration 1001 / 3820) loss: 1.484791
(Iteration 1101 / 3820) loss: 1.449998
(Epoch 3 / 10) train acc: 0.485000; val_acc: 0.478
000
(Iteration 1201 / 3820) loss: 1.591251
(Iteration 1301 / 3820) loss: 1.776905
(Iteration 1401 / 3820) loss: 1.648342
(Iteration 1501 / 3820) loss: 1.645690
(Epoch 4 / 10) train acc: 0.494000; val_acc: 0.511
000
(Iteration 1601 / 3820) loss: 1.765126
(Iteration 1701 / 3820) loss: 1.486973
(Iteration 1801 / 3820) loss: 1.595435
(Iteration 1901 / 3820) loss: 1.499699
(Epoch 5 / 10) train acc: 0.520000; val_acc: 0.517
000
(Iteration 2001 / 3820) loss: 1.515967
(Iteration 2101 / 3820) loss: 1.431810
(Iteration 2201 / 3820) loss: 1.627204
(Epoch 6 / 10) train acc: 0.537000; val_acc: 0.522
000
(Iteration 2301 / 3820) loss: 1.569085
(Iteration 2401 / 3820) loss: 1.573716
(Iteration 2501 / 3820) loss: 1.515708
(Iteration 2601 / 3820) loss: 1.427130
(Epoch 7 / 10) train acc: 0.533000; val_acc: 0.537
000
(Iteration 2701 / 3820) loss: 1.489325
(Iteration 2801 / 3820) loss: 1.495617
(Iteration 2901 / 3820) loss: 1.530800
(Iteration 3001 / 3820) loss: 1.493265
(Epoch 8 / 10) train acc: 0.548000; val_acc: 0.535
000
(Iteration 3101 / 3820) loss: 1.522842
(Iteration 3201 / 3820) loss: 1.489426
(Iteration 3301 / 3820) loss: 1.376905
(Iteration 3401 / 3820) loss: 1.521462
```

```
(Epoch 9 / 10) train acc: 0.553000; val_acc: 0.532
000
(Iteration 3501 / 3820) loss: 1.264398
(Iteration 3601 / 3820) loss: 1.349845
(Iteration 3701 / 3820) loss: 1.331057
(Iteration 3801 / 3820) loss: 1.372419
(Epoch 10 / 10) train acc: 0.597000; val_acc: 0.55
5000
```

Test your model!

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

In [44]:

```
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']).m
ean())
```

```
Validation set accuracy:  0.555
Test set accuracy:  0.538
```

In [ ]:

```

```