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:

```python
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:

```python
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.

In [3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs682.classifiers.fc_net import *
from cs682.data_utils import get_CIFAR10_data
from cs682.gradient_check import eval_numerical_gradient, eval_numerical
_gradient_array
from cs682.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-i
n-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
))))
```

In [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,))
```

Affine layer: foward

Open the file `cs682/layers.py` and implement the `affine_forward` function.

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

In [36]:
```
# 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_sh
ape)
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
```

Affine layer: backward

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

In [37]:

```
# 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
```

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 [38]:

```
# 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,  0.13636
364,],
                        [ 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
```

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 [5]:
```
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:

Sigmoid and ReLU\ Sigmoid has close to zero gradient for very large positive as well as negative values. ReLU has zero gradient for all values<0.

"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 `cs682/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 [40]:
```
from cs682.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
```

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 `cs682/layers.py` .

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

In [41]:
```
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, ver
bose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error s
hould 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 `cs682/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 [42]:

```
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_sca
le=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 los
s'

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

# 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=Fa
lse)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[n
ame])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-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: 3.12e-07
W2 relative error: 7.98e-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.

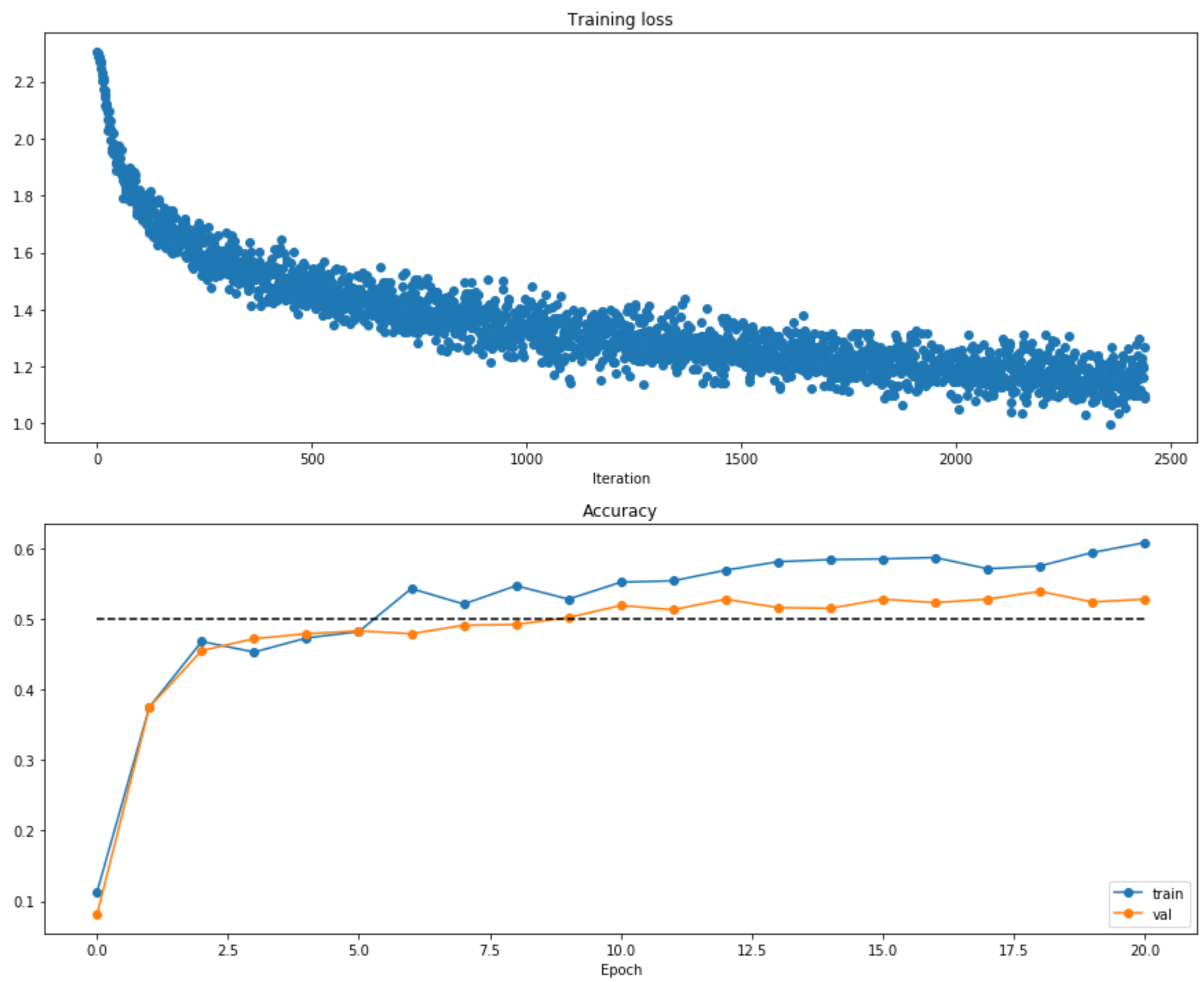
Open the file `cs682/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

In [43]:
```
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at le
ast  #
# 50% accuracy on the validation set.
#
##############################################################################
data = {'X_train':data['X_train'], 'y_train':data['y_train'], 'X_val':da
ta['X_val'], 'y_val':data['y_val']}

solver = Solver(model, data,
                update_rule = 'sgd',
                optim_config = {'learning_rate':1e-3},
                lr_decay = 0.95,
                num_epochs = 20,
                batch_size = 400,
                print_every = 100)

best_params = solver.train()
##############################################################################
#                              END OF YOUR CODE
#
##############################################################################
```

```
(Iteration 1 / 2440) loss: 2.304953
(Epoch 0 / 20) train acc: 0.113000; val_acc: 0.081000
(Iteration 101 / 2440) loss: 1.769195
(Epoch 1 / 20) train acc: 0.375000; val_acc: 0.375000
(Iteration 201 / 2440) loss: 1.669165
(Epoch 2 / 20) train acc: 0.468000; val_acc: 0.455000
(Iteration 301 / 2440) loss: 1.670013
(Epoch 3 / 20) train acc: 0.453000; val_acc: 0.472000
(Iteration 401 / 2440) loss: 1.558850
(Epoch 4 / 20) train acc: 0.473000; val_acc: 0.479000
(Iteration 501 / 2440) loss: 1.489412
(Iteration 601 / 2440) loss: 1.365843
(Epoch 5 / 20) train acc: 0.482000; val_acc: 0.483000
(Iteration 701 / 2440) loss: 1.411090
(Epoch 6 / 20) train acc: 0.543000; val_acc: 0.479000
(Iteration 801 / 2440) loss: 1.260651
(Epoch 7 / 20) train acc: 0.521000; val_acc: 0.491000
(Iteration 901 / 2440) loss: 1.318146
(Epoch 8 / 20) train acc: 0.547000; val_acc: 0.492000
(Iteration 1001 / 2440) loss: 1.344013
(Epoch 9 / 20) train acc: 0.528000; val_acc: 0.502000
(Iteration 1101 / 2440) loss: 1.342403
(Iteration 1201 / 2440) loss: 1.257451
(Epoch 10 / 20) train acc: 0.552000; val_acc: 0.519000
(Iteration 1301 / 2440) loss: 1.314607
(Epoch 11 / 20) train acc: 0.554000; val_acc: 0.513000
(Iteration 1401 / 2440) loss: 1.260950
(Epoch 12 / 20) train acc: 0.569000; val_acc: 0.528000
(Iteration 1501 / 2440) loss: 1.294013
(Epoch 13 / 20) train acc: 0.581000; val_acc: 0.516000
(Iteration 1601 / 2440) loss: 1.206977
(Iteration 1701 / 2440) loss: 1.250007
(Epoch 14 / 20) train acc: 0.584000; val_acc: 0.515000
(Iteration 1801 / 2440) loss: 1.285593
(Epoch 15 / 20) train acc: 0.585000; val_acc: 0.528000
(Iteration 1901 / 2440) loss: 1.309243
(Epoch 16 / 20) train acc: 0.587000; val_acc: 0.523000
(Iteration 2001 / 2440) loss: 1.150673
(Epoch 17 / 20) train acc: 0.571000; val_acc: 0.528000
(Iteration 2101 / 2440) loss: 1.183055
(Epoch 18 / 20) train acc: 0.575000; val_acc: 0.539000
(Iteration 2201 / 2440) loss: 1.144189
(Iteration 2301 / 2440) loss: 1.186106
(Epoch 19 / 20) train acc: 0.594000; val_acc: 0.524000
(Iteration 2401 / 2440) loss: 1.162940
(Epoch 20 / 20) train acc: 0.608000; val_acc: 0.528000
```

In [44]:

```
# 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()
```



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 `cs682/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 [45]:

```python
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=Fa
lse, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[n
ame])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.90e-09
W2 relative error: 6.87e-08
W3 relative error: 2.13e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.57e-10
```

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

In [46]:

```
# 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'],
}

# for trial in range(20):
#     weight_scale = 10**np.random.uniform(-10, 2)
#     learning_rate = 10**np.random.uniform(-10, 2)
#     print(weight_scale, learning_rate)
# Result of random search : weight_scale=0.0620063159371343 learning_rat
e=0.0007450699140127562
weight_scale = 0.06
learning_rate = 0.0007
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()
```

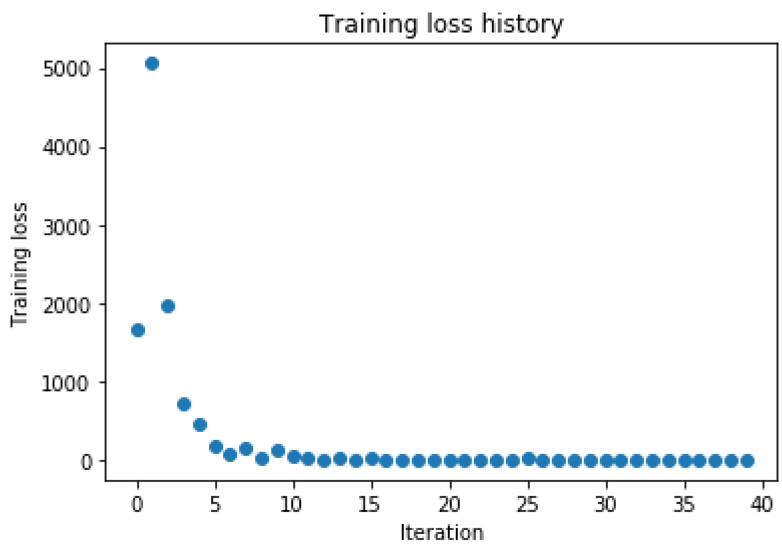
```
(Iteration 1 / 40) loss: 77.236010
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.121000
(Epoch 1 / 20) train acc: 0.240000; val_acc: 0.140000
(Epoch 2 / 20) train acc: 0.460000; val_acc: 0.139000
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.157000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.170000
(Epoch 5 / 20) train acc: 0.900000; val_acc: 0.184000
(Iteration 11 / 40) loss: 0.012121
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.184000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.181000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.176000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.178000
(Epoch 10 / 20) train acc: 0.920000; val_acc: 0.167000
(Iteration 21 / 40) loss: 2.035583
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.171000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.165000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.165000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.165000
(Iteration 31 / 40) loss: 0.000103
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.165000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.166000
```

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

In [7]:

```
# 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'],
}

# for trial in range(20):
#     weight_scale = 10**np.random.uniform(-5, 2)
#     learning_rate = 10**np.random.uniform(-5, 2)
#     print("weight_scale, learning_rate)
# Result of random search: 0.17227070584921386 0.00045953640384386875
learning_rate = 0.0004
weight_scale = 0.17
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=1000, 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: 1660.237200
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.105000
(Epoch 1 / 20) train acc: 0.140000; val_acc: 0.115000
(Epoch 2 / 20) train acc: 0.340000; val_acc: 0.105000
(Epoch 3 / 20) train acc: 0.520000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.740000; val_acc: 0.146000
(Epoch 5 / 20) train acc: 0.820000; val_acc: 0.156000
(Epoch 6 / 20) train acc: 0.860000; val_acc: 0.147000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.152000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.152000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.151000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.145000
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.145000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.145000
(Epoch 13 / 20) train acc: 0.940000; val_acc: 0.155000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.150000
(Epoch 15 / 20) train acc: 0.960000; val_acc: 0.141000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.142000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.142000
```



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:

The five layer net was more difficult to overfit and it was also much more sensitive to the initialization scale. Having a network with more layers makes it susceptible to gradients becoming to large (exploding gradients) or too small (vanishing gradients) which running backpropogation. This increases the difficulty as the network does not learn anything when this happens. With a deeper network, the impact of the initial values of the weights gets compounded and hence, the deeper network becomes more sensitive to the initial weight values.

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 [https://compsci682-fa19.github.io/notes/neural-networks-3/#sgd (https://compsci682-fa19.github.io/notes/neural-networks-3/#sgd)](https://compsci682-fa19.github.io/notes/neural-networks-3/#sgd) for more information.

Open the file `cs682/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 [48]:
```
from cs682.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'
]))
```

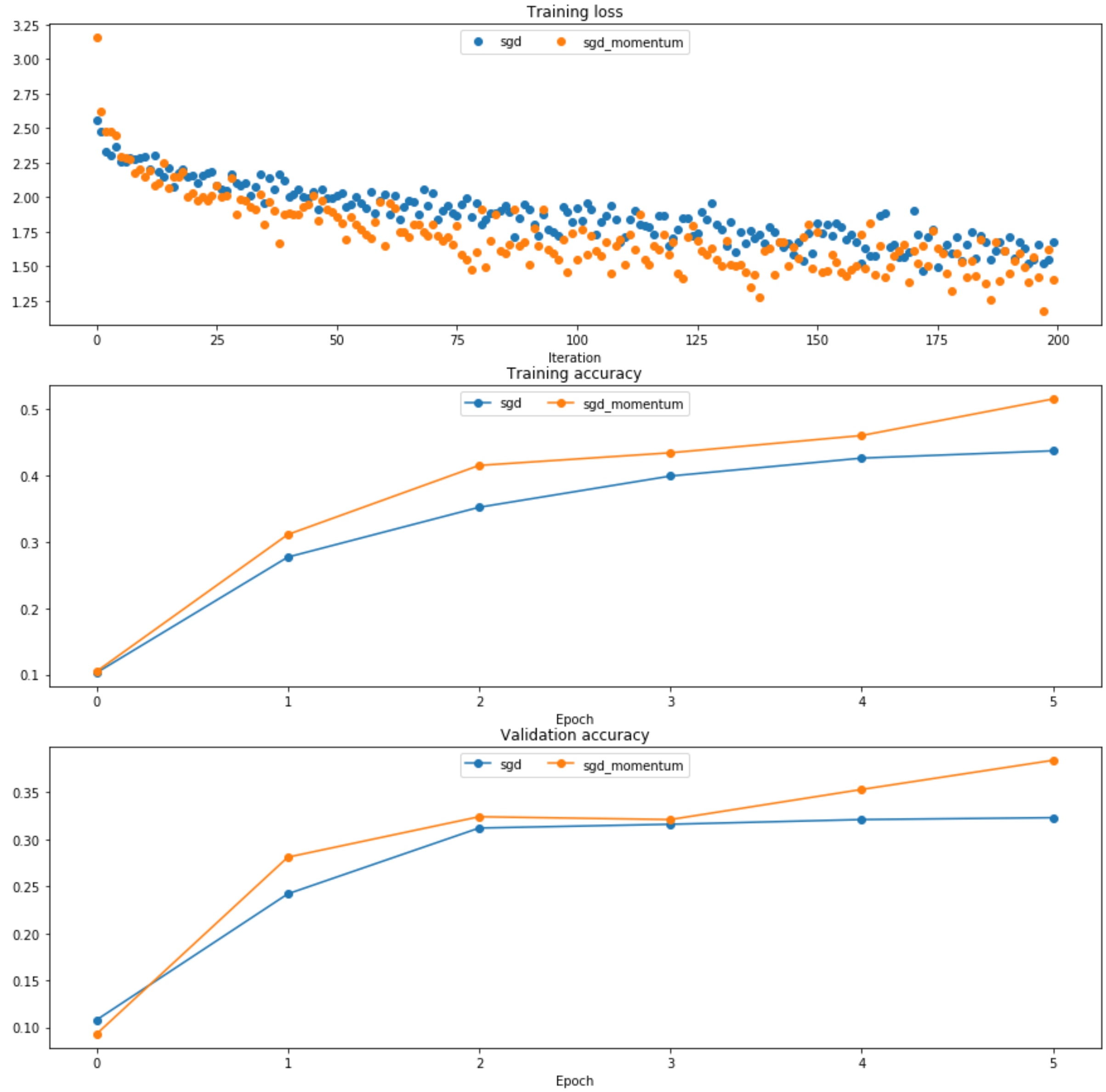
```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

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.

In [49]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

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

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

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

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

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

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

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

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

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

```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082693
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.023753
(Iteration 71 / 200) loss: 2.026621
(Epoch 2 / 5) train acc: 0.352000; val_acc: 0.312000
(Iteration 81 / 200) loss: 1.807163
(Iteration 91 / 200) loss: 1.914256
(Iteration 101 / 200) loss: 1.920494
(Iteration 111 / 200) loss: 1.708877
(Epoch 3 / 5) train acc: 0.399000; val_acc: 0.316000
(Iteration 121 / 200) loss: 1.701111
(Iteration 131 / 200) loss: 1.769697
(Iteration 141 / 200) loss: 1.788899
(Iteration 151 / 200) loss: 1.816437
(Epoch 4 / 5) train acc: 0.426000; val_acc: 0.321000
(Iteration 161 / 200) loss: 1.633853
(Iteration 171 / 200) loss: 1.903011
(Iteration 181 / 200) loss: 1.540134
(Iteration 191 / 200) loss: 1.712615
(Epoch 5 / 5) train acc: 0.437000; val_acc: 0.323000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032562
(Iteration 31 / 200) loss: 1.985848
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882354
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512046
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610417
(Iteration 181 / 200) loss: 1.528331
(Iteration 191 / 200) loss: 1.447238
(Epoch 5 / 5) train acc: 0.515000; val_acc: 0.384000
```

```
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:39: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:42: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:45: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:49: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each 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 `cs682/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 [50]:
```python
# Test RMSProp implementation
from cs682.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
```

In [51]:

```python
# Test Adam implementation
from cs682.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']))
```

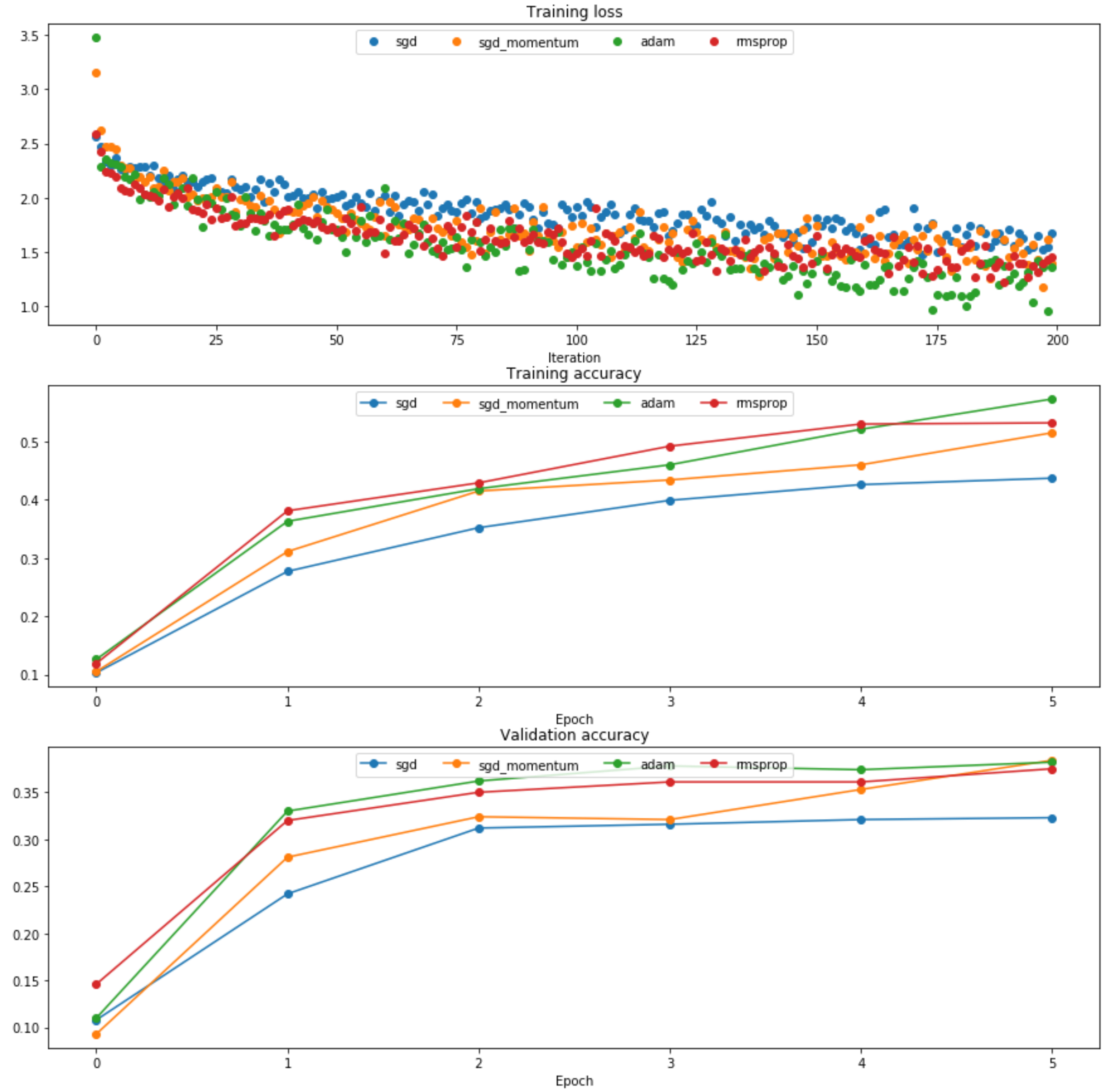
```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

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

In [52]:
```
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)
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: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.027712
(Iteration 21 / 200) loss: 2.183358
(Iteration 31 / 200) loss: 1.744257
(Epoch 1 / 5) train acc: 0.363000; val_acc: 0.330000
(Iteration 41 / 200) loss: 1.707951
(Iteration 51 / 200) loss: 1.703835
(Iteration 61 / 200) loss: 2.094758
(Iteration 71 / 200) loss: 1.505558
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.594429
(Iteration 91 / 200) loss: 1.519017
(Iteration 101 / 200) loss: 1.368522
(Iteration 111 / 200) loss: 1.470400
(Epoch 3 / 5) train acc: 0.460000; val_acc: 0.378000
(Iteration 121 / 200) loss: 1.199064
(Iteration 131 / 200) loss: 1.464705
(Iteration 141 / 200) loss: 1.359863
(Iteration 151 / 200) loss: 1.415069
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.374000
(Iteration 161 / 200) loss: 1.382818
(Iteration 171 / 200) loss: 1.359900
(Iteration 181 / 200) loss: 1.095947
(Iteration 191 / 200) loss: 1.243088
(Epoch 5 / 5) train acc: 0.573000; val_acc: 0.382000

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: 1.487204
(Iteration 71 / 200) loss: 1.629973
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.506686
(Iteration 91 / 200) loss: 1.610742
(Iteration 101 / 200) loss: 1.486124
(Iteration 111 / 200) loss: 1.559454
(Epoch 3 / 5) train acc: 0.492000; val_acc: 0.361000
(Iteration 121 / 200) loss: 1.497406
(Iteration 131 / 200) loss: 1.530736
(Iteration 141 / 200) loss: 1.550957
(Iteration 151 / 200) loss: 1.652046
(Epoch 4 / 5) train acc: 0.530000; val_acc: 0.361000
(Iteration 161 / 200) loss: 1.599574
(Iteration 171 / 200) loss: 1.401073
(Iteration 181 / 200) loss: 1.509582
(Iteration 191 / 200) loss: 1.368611
(Epoch 5 / 5) train acc: 0.532000; val_acc: 0.375000
```

```
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:30: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:33: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:36: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
/Users/anshuman/anaconda3/envs/cs682/lib/python3.6/site-packages/ipyker
nel_launcher.py:40: MatplotlibDeprecationWarning: Adding an axes using
the same arguments as a previous axes currently reuses the earlier inst
ance.  In a future version, a new instance will always be created and r
eturned.  Meanwhile, this warning can be suppressed, and the future beh
avior ensured, by passing a unique label to each axes instance.
```



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:

Adagrad accumulates the gradient and divides by it to compute the updates. This value can become too large after several iterations after which, the updates would become extremely small. Adam, however, takes a weighted average of the accumulated gradient (cache) and the current gradient (dw), hence it would not explode in as many iterations. Thus, Adam is more robust to this issue over several iterations of updates. Consider an example where the gradients are 10 at each step. With AdaGrad, the cache would become 1000 in 100 iterations whereas with Adam, the cache would remain 10. More formally, the Adam cache would never be greater than the max gradient. Thus, Adam does not suffer from this issue.

Alternatively, if the network suffers from exploding gradients (gradients becoming too large), then division by the accumulated value (in cache) could lead to a very small update (in very few iterations). The Adam update takes a weighted average, hence the updates would not be impacted by this.

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 [131]:

```
best_model = None
################################################################################
# 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.                                                     #
################################################################################
numTrials = 25
results = []
bestHyperparams = None
np.random.seed(123)
learning_rates = 10**np.random.uniform(-2,-4, numTrials)
regs = 10**np.random.uniform(0,-4, numTrials)
# dropouts = np.random.uniform(0.6, 1, numTrials)
weight_scales = np.random.uniform(1e-2,5e-2, numTrials)
for trial in range (numTrials):
    learning_rate = learning_rates[trial]
    reg = regs[trial]
    weight_scale = weight_scales[trial]
    dropout = 1#dropouts[trial]

    print ("Trial {}: lr={} reg={} weight_scale={} do={}".format(trial, learning_rate, reg, weight_scale, dropout))
    model = FullyConnectedNet([100, 400, 400, 400, 100],
                              weight_scale=weight_scale,
                              normalization='batchnorm',
                              dtype=np.float64,
                              dropout=dropout,
                              reg=reg,
                              seed=123)

    solver = Solver(model, data,
                    num_epochs=20,
                    batch_size=50,
                    update_rule='adam',
                    optim_config={'learning_rate': learning_rate},
                    verbose=False,
                    lr_decay = 0.95,
                    print_every = 100)

    solver.train()
    val_acc = solver.best_val_acc
    print ("Best Train acc = {} Val acc = {}".format(max(solver.train_acc_history), val_acc))
#     print ("Min loss =", min(solver.loss_history))
#     print(solver.val_acc_history)
    if bestHyperparams is None or bestHyperparams[4] < val_acc:
        bestHyperparams = (learning_rate, reg, weight_scale, dropout, val_acc)
        best_model = model
    results.append((learning_rate, reg, weight_scale, dropout, val_acc))
```

```
print("Best hyperparams", bestHyperparams)
##########################################################################
########
#                                   END OF YOUR CODE
#
##########################################################################
########
```

```
Trial 0: lr=0.0004046333073420584 reg=0.05106982205052503 6 weight_scale
=0.0148251466396129 5 do=1
Best Train acc = 0.522 Val acc = 0.519
Trial 1: lr=0.0026774497579933826 reg=0.035714566043029326 weight_scale
=0.0430536320202733 3 do=1
Best Train acc = 0.372 Val acc = 0.362
Trial 2: lr=0.003518010188412758 7 reg=0.122165077963191 5 weight_scale=
0.034122405136437 1 do=1
Best Train acc = 0.279 Val acc = 0.281
Trial 3: lr=0.000789533318080700 5 reg=0.06685652729611 9 weight_scale=0.
031802720258658594 do=1
Best Train acc = 0.47 Val acc = 0.446
Trial 4: lr=0.000363967043090007 reg=0.002992922730182325 weight_scale=
0.023710553350972337 do=1
Best Train acc = 0.774 Val acc = 0.581
Trial 5: lr=0.0014249088365594447 reg=0.42813451438371275 weight_scale=
0.022164831561087363 do=1
Best Train acc = 0.262 Val acc = 0.274
Trial 6: lr=0.00010926261824221895 reg=0.018416000127050833 weight_scal
e=0.026680888440988065 do=1
Best Train acc = 0.826 Val acc = 0.566
Trial 7: lr=0.00042691412362385033 reg=0.018903792722959698 weight_scal
e=0.03725203063171187 do=1
Best Train acc = 0.584 Val acc = 0.529
Trial 8: lr=0.0010917826715488564 reg=0.010598871074605358 weight_scale
=0.045018273671806996 do=1
Best Train acc = 0.529 Val acc = 0.492
Trial 9: lr=0.0016434820443355922 reg=0.019800622263578854 weight_scale
=0.030416893499120447 do=1
Best Train acc = 0.465 Val acc = 0.445
Trial 10: lr=0.0020589413120966965 reg=0.05635793970800434 weight_scale
=0.036772551318490894 do=1
Best Train acc = 0.385 Val acc = 0.388
Trial 11: lr=0.0003482575859315551 reg=0.01970583182808769 weight_scale
=0.03343746210248852 do=1
Best Train acc = 0.62 Val acc = 0.546
Trial 12: lr=0.0013269529083173167 reg=0.0002669582505040336 weight_sca
le=0.034996140083824 do=1
Best Train acc = 0.751 Val acc = 0.56
Trial 13: lr=0.007597036391571568 reg=0.00016724761263818577 weight_sca
le=0.03698756203951299 do=1
Best Train acc = 0.534 Val acc = 0.51
Trial 14: lr=0.001599232066581871 reg=0.009832258690034188 weight_scale
=0.043693697504810294 do=1
Best Train acc = 0.501 Val acc = 0.481
Trial 15: lr=0.0003342021107943263 reg=0.003192921143915423 weight_scal
e=0.013327799533297552 do=1
Best Train acc = 0.77 Val acc = 0.567
Trial 16: lr=0.004315355105388501 reg=0.3447675793771386 weight_scale=
0.04054731365773353 do=1
Best Train acc = 0.199 Val acc = 0.22
Trial 17: lr=0.004457552713588758 reg=0.05380939048839085 weight_scale=
0.019746654981474 96 do=1
Best Train acc = 0.281 Val acc = 0.295
Trial 18: lr=0.0008647633042996515 reg=0.021912662622563427 weight_scal
e=0.017768918423150835 do=1
Best Train acc = 0.502 Val acc = 0.481
```

```
Trial 19: lr=0.0008636640168530538 reg=0.00034258107550713153 weight_sc
ale=0.032898278299658926 do=1
Best Train acc = 0.793 Val acc = 0.567
Trial 20: lr=0.0005385153448283213 reg=0.09958147125344827 weight_scale
=0.013828500664495485 do=1
Best Train acc = 0.463 Val acc = 0.462
Trial 21: lr=0.0002000490124667205 reg=0.011691303721755232 weight_scal
e=0.04541307305100559 do=1
Best Train acc = 0.766 Val acc = 0.556
Trial 22: lr=0.00035570449248140194 reg=0.00011422491868421907 weight_s
cale=0.03508995888205075 do=1
Best Train acc = 0.903 Val acc = 0.55
Trial 23: lr=0.0005997261399883596 reg=0.008357175510570808 weight_scal
e=0.03893665432759819 do=1
Best Train acc = 0.613 Val acc = 0.554
Trial 24: lr=0.00035901552960287584 reg=0.003535264375693929 weight_sca
le=0.010645168267800674 do=1
Best Train acc = 0.728 Val acc = 0.557
Best hyperparams (0.000363967043090007, 0.002992922730182325, 0.0237105
53350972337, 1, 0.581)
```

Test your model!

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

In [135]:

```
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:  0.551
Test set accuracy:  0.507
```

In [ ]: