FullyConnectedNets

October 20, 2019

1 Fully-Connected Neural Nets

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

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

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

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

  return out, cache
```

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

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

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

```
  dw = # Derivative of loss with respect to w

  return dx, dw
```

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

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

```
In [1]: # 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-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))))

In [2]: # Load the (preprocessed) CIFAR10 data.

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

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

2 Affine layer: foward

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

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

```
In [3]: # Test the affine_forward function

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

        input_size = num_inputs * np.prod(input_shape)
```

```
        weight_size = output_dim * np.prod(input_shape)

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

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

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

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

3 Affine layer: backward

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

In [4]:
```
        # Test the affine_backward function
        np.random.seed(231)
        x = np.random.randn(10, 2, 3)
        w = np.random.randn(6, 5)
        b = np.random.randn(5)
        dout = np.random.randn(10, 5)

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

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

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

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

4 ReLU activation: forward

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

In [5]:
```
        # Test the relu_forward function

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

        out, _ = relu_forward(x)
        correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
```

```
                                [ 0.,          0.,          0.04545455,  0.13636364,],
                                [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

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

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

5 ReLU activation: backward

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

```
In [6]: np.random.seed(231)
        x = np.random.randn(10, 10)
        dout = np.random.randn(*x.shape)

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

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

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

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

5.1 Inline Question 1:

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

5.2 Answer:

Sigmoid and ReLU are both capable of producing zero gradient flow during backpropagation.

Sigmoid will have zero gradient if the input is too large in magnitude (either potitive or negative). In the one-dimensional case, the gradient would be zero if the input were a large positive or negative number.

ReLU will have zero gradient if the input is negative. In the one-dimensional case, the gradient would be zero if the input were a negative number.

Leaky ReLU could also potentially produce *close to zero* gradient flow during backpropagation if the slope of the "leaky part" isn't high enough (i.e. if the small positive constant that negative inputs are multiplied by is *too* small).

6 "Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `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 [7]: 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:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs682/layers.py`.

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

```
In [8]: np.random.seed(231)
        num_classes, num_inputs = 10, 50
        x = 0.001 * np.random.randn(num_inputs, num_classes)
        y = np.random.randint(num_classes, size=num_inputs)

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

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

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

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

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

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

8 Two-layer network

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

Open the file `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 [9]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,
16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,
16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,
16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'
```

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

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

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

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

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

9 Solver

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

Open the file `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.

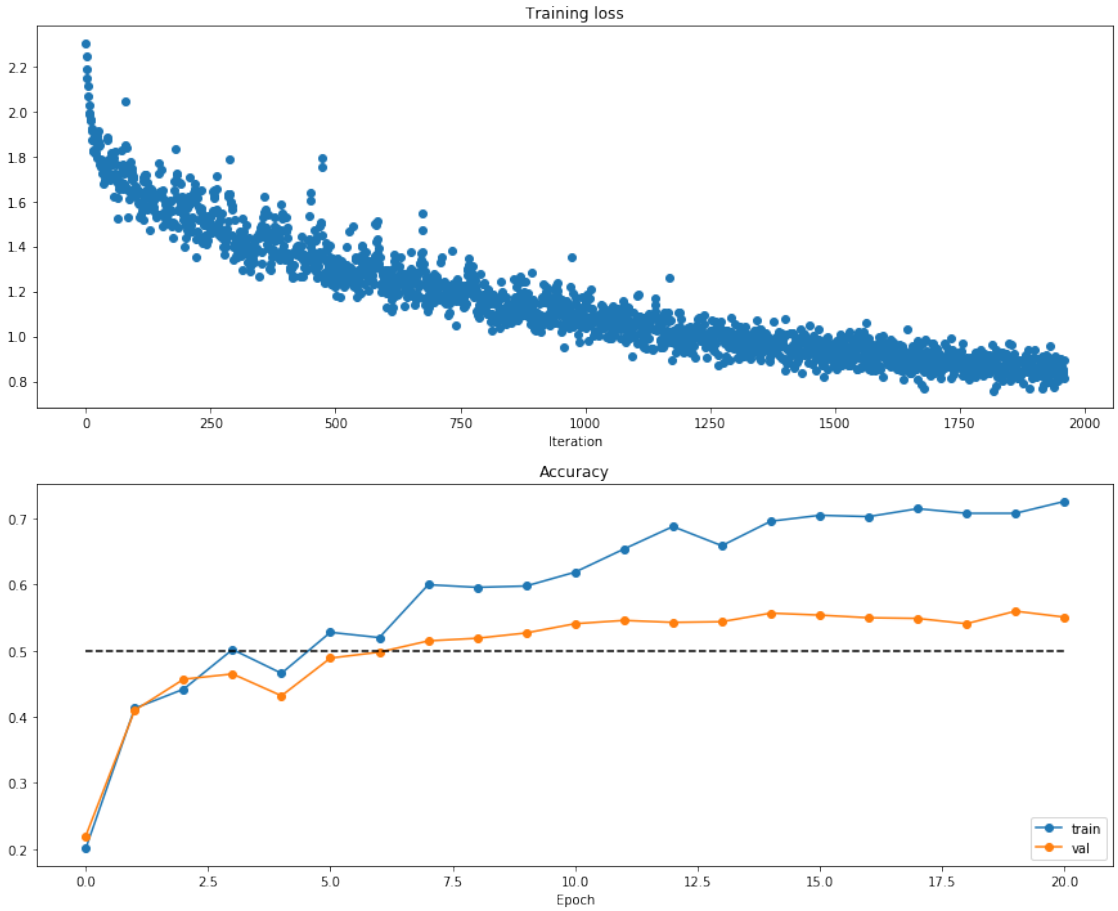
In [10]:
```
        ################################################################################
        # TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
        # 50% accuracy on the validation set.                                          #
        ################################################################################

        solver_data = {'X_train': data['X_train'],
                       'y_train': data['y_train'],
                       'X_val': data['X_val'],
                       'y_val': data['y_val']
                      }

        def hyperparameter_search():
            learning_rates = np.arange(1e-3, 5e-3, 5e-4)
            reg_strengths = np.arange(1e-5, 1e-4, 5e-6)
            lr_decays = np.arange(0.75, 0.95, 0.01)
```

```
    best_acc = -np.Infinity
    best_model = None
    best_params = None

    while best_acc < 0.5:
        learning_rate = np.random.choice(learning_rates)
        reg_strength = np.random.choice(reg_strengths)
        lr_decay = np.random.choice(lr_decays)
        optim_config = {'learning_rate': learning_rate}

        print('learning rate = %e, reg strength = %e, lr decay = %f' % (learning_rate,
reg_strength, lr_decay))

        # preselected parameters based on manual tuning
        batch_size = 500
        hidden_dim = 500
        num_epochs = 20

        model = TwoLayerNet(hidden_dim=hidden_dim,
                            reg=reg_strength)

        solver = Solver(model,
                        solver_data,
                        optim_config=optim_config,
                        lr_decay=lr_decay,
                        batch_size=batch_size,
                        num_epochs=num_epochs,
                        print_every=np.Infinity)

        solver.train()

        test_acc = solver.check_accuracy(data['X_test'], data['y_test'])
        print('test set accuracy = ', test_acc)

        if test_acc > best_acc:
            print('----- updating parameters -----')
            best_params = tuple([learning_rate, reg_strength, lr_decay])
            best_acc = test_acc
            best_model = model

    return best_model, best_params

# By default, the notebook will run the parameters found on a previous run of the
hyperparameter_search()
# function defined above, which resulted in a test accuracy of 55.8%. If you want to
perform hyperparameter
# search, change the value of the following variable to True
do_hyperparameter_search = False

if do_hyperparameter_search:
    model, params = hyperparameter_search()
else:
    # parameters found by running hyperparameter_search() resulting in a test accuracy
of 55.8%
    optim_config = {'learning_rate': 3e-3}
    reg_strength = 5e-5
    lr_decay = 0.9
    hidden_dim = 500
    batch_size=500
    num_epochs = 20

    model = TwoLayerNet(hidden_dim=500, reg=reg_strength)
    solver = Solver(model,
                    solver_data,
                    optim_config=optim_config,
                    lr_decay=lr_decay,
                    batch_size=500,
```

```
                        num_epochs=20,
                        print_every=np.Infinity)
            solver.train()

            test_acc = solver.check_accuracy(data['X_test'], data['y_test'])
            print('test set accuracy = ', test_acc)

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

```
(Iteration 1 / 1960) loss: 2.311008
(Epoch 0 / 20) train acc: 0.211000; val_acc: 0.251000
(Epoch 1 / 20) train acc: 0.419000; val_acc: 0.407000
(Epoch 2 / 20) train acc: 0.426000; val_acc: 0.428000
(Epoch 3 / 20) train acc: 0.457000; val_acc: 0.426000
(Epoch 4 / 20) train acc: 0.501000; val_acc: 0.456000
(Epoch 5 / 20) train acc: 0.500000; val_acc: 0.461000
(Epoch 6 / 20) train acc: 0.554000; val_acc: 0.500000
(Epoch 7 / 20) train acc: 0.559000; val_acc: 0.510000
(Epoch 8 / 20) train acc: 0.632000; val_acc: 0.539000
(Epoch 9 / 20) train acc: 0.606000; val_acc: 0.527000
(Epoch 10 / 20) train acc: 0.633000; val_acc: 0.521000
(Epoch 11 / 20) train acc: 0.646000; val_acc: 0.546000
(Epoch 12 / 20) train acc: 0.664000; val_acc: 0.560000
(Epoch 13 / 20) train acc: 0.646000; val_acc: 0.532000
(Epoch 14 / 20) train acc: 0.695000; val_acc: 0.552000
(Epoch 15 / 20) train acc: 0.688000; val_acc: 0.555000
(Epoch 16 / 20) train acc: 0.709000; val_acc: 0.555000
(Epoch 17 / 20) train acc: 0.737000; val_acc: 0.555000
(Epoch 18 / 20) train acc: 0.705000; val_acc: 0.563000
(Epoch 19 / 20) train acc: 0.719000; val_acc: 0.564000
(Epoch 20 / 20) train acc: 0.739000; val_acc: 0.555000
test set accuracy =  0.546
```

In [45]:
```
# Run this cell to visualize training loss and train / val accuracy

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

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



10 Multilayer network

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

Read through the `FullyConnectedNet` class in the file `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.

10.1 Initial loss and gradient check

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

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

```
In [106]: np.random.seed(231)
          N, D, H1, H2, C = 2, 15, 20, 30, 10
          X = np.random.randn(N, D)
          y = np.random.randint(C, size=(N,))

          for reg in [0, 3.14]:
```

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

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

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

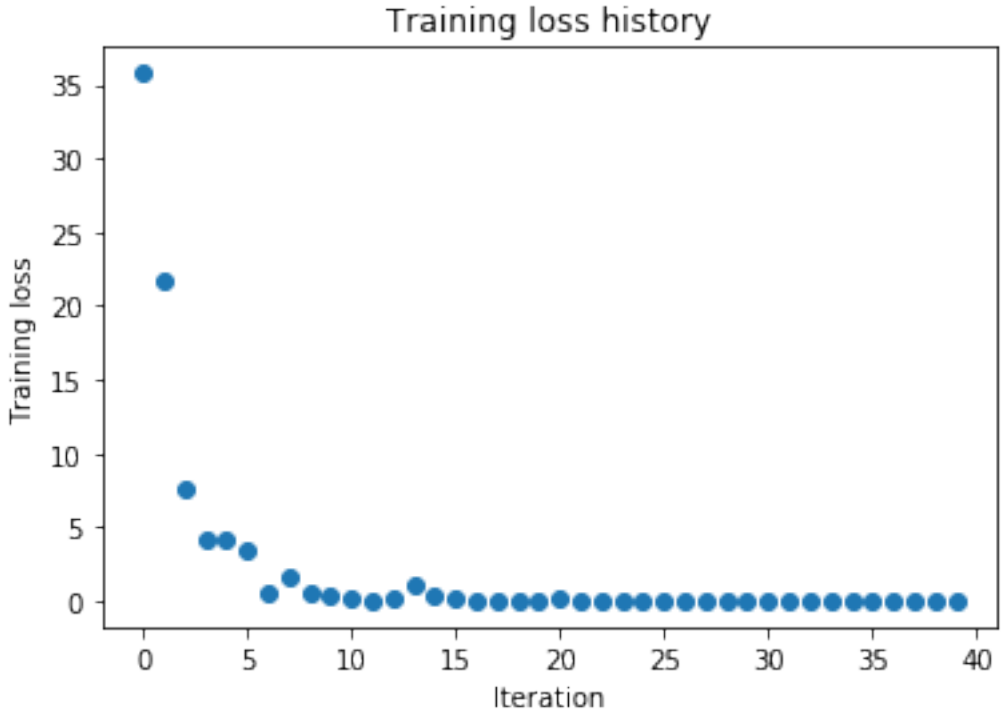
```
Running check with reg =  0
Initial loss:  2.2658787769555633
W1 relative error: 1.81e-08
W2 relative error: 3.73e-07
W3 relative error: 5.55e-07
b1 relative error: 3.13e-10
b2 relative error: 4.67e-11
b3 relative error: 8.06e-11
Running check with reg =  3.14
Initial loss:  450.6109940039681
W1 relative error: 2.57e-07
W2 relative error: 1.69e-06
W3 relative error: 4.11e-08
b1 relative error: 2.68e-08
b2 relative error: 5.54e-09
b3 relative error: 7.75e-09
```

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 [107]:
```
# 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-2
learning_rate = 1e-4
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: 35.790877
(Epoch 0 / 20) train acc: 0.180000; val_acc: 0.119000
(Epoch 1 / 20) train acc: 0.340000; val_acc: 0.120000
(Epoch 2 / 20) train acc: 0.640000; val_acc: 0.140000
(Epoch 3 / 20) train acc: 0.840000; val_acc: 0.157000
(Epoch 4 / 20) train acc: 0.820000; val_acc: 0.150000
(Epoch 5 / 20) train acc: 0.900000; val_acc: 0.147000
(Iteration 11 / 40) loss: 0.100810
(Epoch 6 / 20) train acc: 0.920000; val_acc: 0.154000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.170000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.158000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.158000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.158000
(Iteration 21 / 40) loss: 0.121191
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.160000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.159000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.159000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.158000
(Iteration 31 / 40) loss: 0.000467
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.158000
```

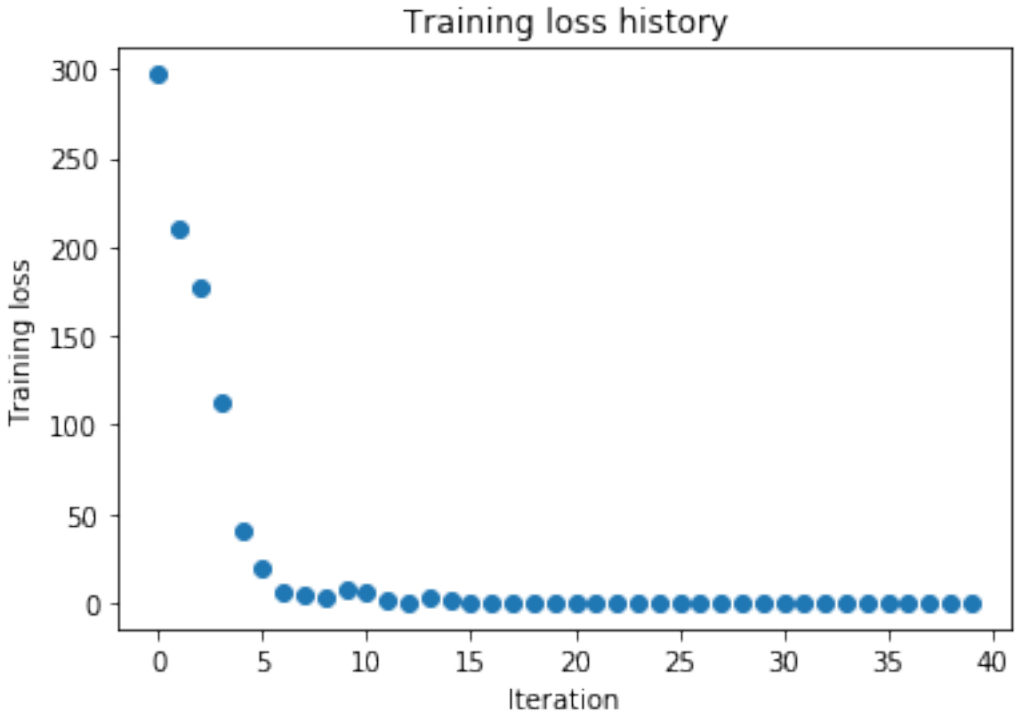


Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [108]:
```
# 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'],
}

weight_scale = 6.5e-2
learning_rate = 3.5e-4
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: 297.208644
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.116000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.106000
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.105000
(Epoch 3 / 20) train acc: 0.660000; val_acc: 0.114000
(Epoch 4 / 20) train acc: 0.780000; val_acc: 0.103000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.120000
(Iteration 11 / 40) loss: 5.387502
(Epoch 6 / 20) train acc: 0.920000; val_acc: 0.121000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.123000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.124000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.129000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.129000
(Iteration 21 / 40) loss: 0.419520
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.125000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.129000
(Iteration 31 / 40) loss: 0.000006
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.129000
```

```
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.129000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.129000
```



10.2 Inline Question 2:

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

10.3 Answer:

The five layer net was more difficult to train, as it was more sensitive to both weight initialization and changes in the learning rate. Smaller initialization weights result in lower gradient flow through ReLU nonlinearities, and because a deeper network has more ReLU layers, there is a higer probability of "dead" ReLU units, as activations become smaller after passing through each affine layer, and are more likely to become zero after passing through each ReLU layer.

11 Update rules

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

12 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at 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 [11]: 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.

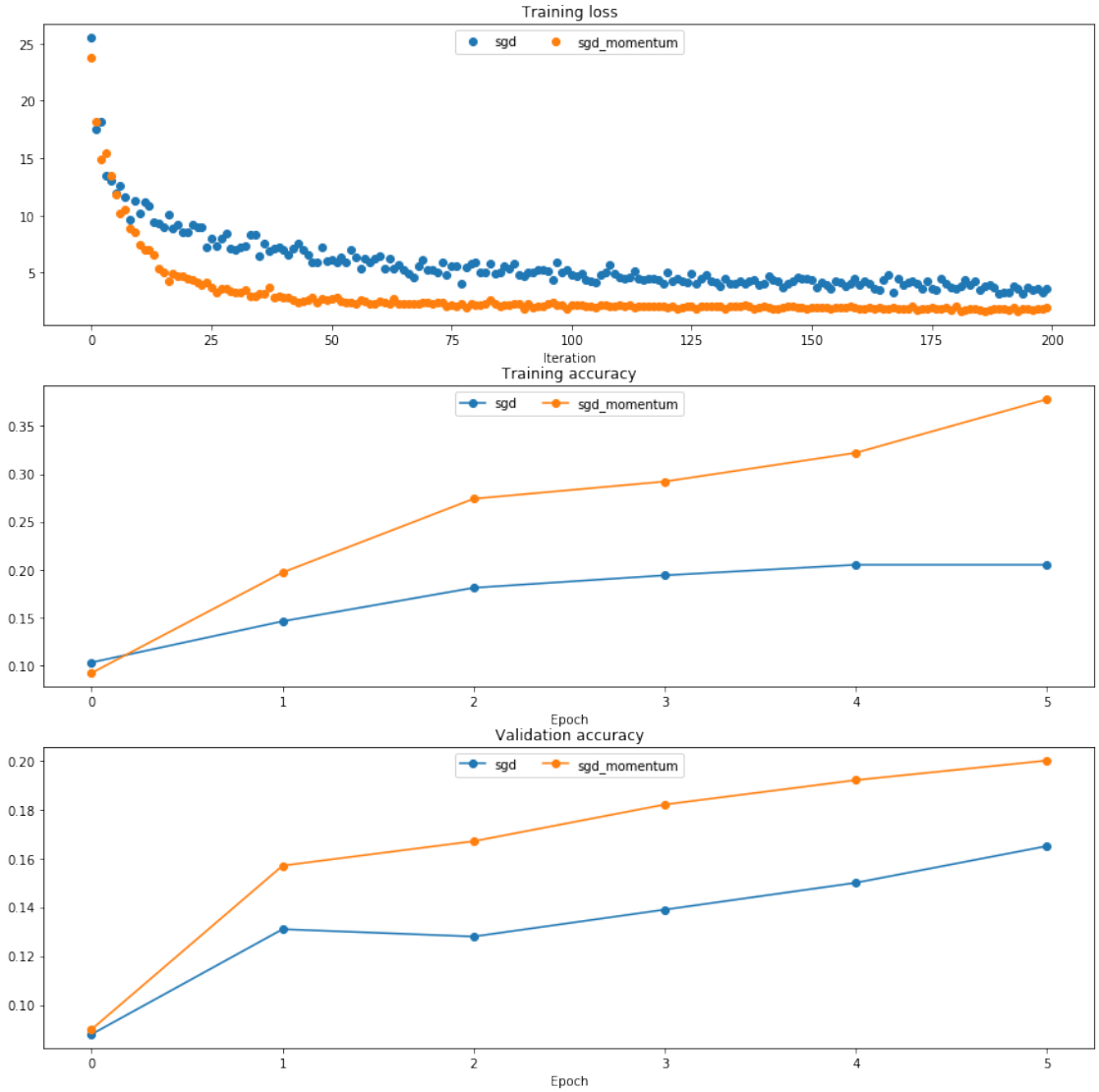
```
In [115]: num_train = 4000
          small_data = {
            'X_train': data['X_train'][:num_train],
            'y_train': data['y_train'][:num_train],
            'X_val': data['X_val'],
            'y_val': data['y_val'],
          }

          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-4,
                                            }
```

```
                        )

            solver.train()
            solvers[update_rule] = solver

            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: 25.495220
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.088000
(Iteration 11 / 200) loss: 10.229651
(Iteration 21 / 200) loss: 8.493636
(Iteration 31 / 200) loss: 6.999047
(Epoch 1 / 5) train acc: 0.146000; val_acc: 0.131000
(Iteration 41 / 200) loss: 7.036191
(Iteration 51 / 200) loss: 6.139806
(Iteration 61 / 200) loss: 6.453247
(Iteration 71 / 200) loss: 5.219362
(Epoch 2 / 5) train acc: 0.181000; val_acc: 0.128000
(Iteration 81 / 200) loss: 5.961541
(Iteration 91 / 200) loss: 4.688865
(Iteration 101 / 200) loss: 4.820385
(Iteration 111 / 200) loss: 4.603017
(Epoch 3 / 5) train acc: 0.194000; val_acc: 0.139000
(Iteration 121 / 200) loss: 5.014280
(Iteration 131 / 200) loss: 4.214131
(Iteration 141 / 200) loss: 4.028232
(Iteration 151 / 200) loss: 4.427943
(Epoch 4 / 5) train acc: 0.205000; val_acc: 0.150000
(Iteration 161 / 200) loss: 3.957780
(Iteration 171 / 200) loss: 4.123300
(Iteration 181 / 200) loss: 3.588122
(Iteration 191 / 200) loss: 3.231773
(Epoch 5 / 5) train acc: 0.205000; val_acc: 0.165000

running with  sgd_momentum
```

```
(Iteration 1 / 200) loss: 23.816016
(Epoch 0 / 5) train acc: 0.092000; val_acc: 0.090000
(Iteration 11 / 200) loss: 7.469200
(Iteration 21 / 200) loss: 4.539695
(Iteration 31 / 200) loss: 3.297007
(Epoch 1 / 5) train acc: 0.197000; val_acc: 0.157000
(Iteration 41 / 200) loss: 2.896729
(Iteration 51 / 200) loss: 2.685854
(Iteration 61 / 200) loss: 2.481272
(Iteration 71 / 200) loss: 2.445718
(Epoch 2 / 5) train acc: 0.274000; val_acc: 0.167000
(Iteration 81 / 200) loss: 2.150510
(Iteration 91 / 200) loss: 1.867980
(Iteration 101 / 200) loss: 2.174606
(Iteration 111 / 200) loss: 2.172702
(Epoch 3 / 5) train acc: 0.292000; val_acc: 0.182000
(Iteration 121 / 200) loss: 1.953552
(Iteration 131 / 200) loss: 2.079026
(Iteration 141 / 200) loss: 2.056502
(Iteration 151 / 200) loss: 1.943504
(Epoch 4 / 5) train acc: 0.322000; val_acc: 0.192000
(Iteration 161 / 200) loss: 1.881984
(Iteration 171 / 200) loss: 1.889364
(Iteration 181 / 200) loss: 2.086546
(Iteration 191 / 200) loss: 1.856802
(Epoch 5 / 5) train acc: 0.378000; val_acc: 0.200000
```

```
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:42:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:45:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:48:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:52:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
```



13 RMSProp and Adam

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

In the file `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 [12]:
```
# 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 [13]:
```
# 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
```

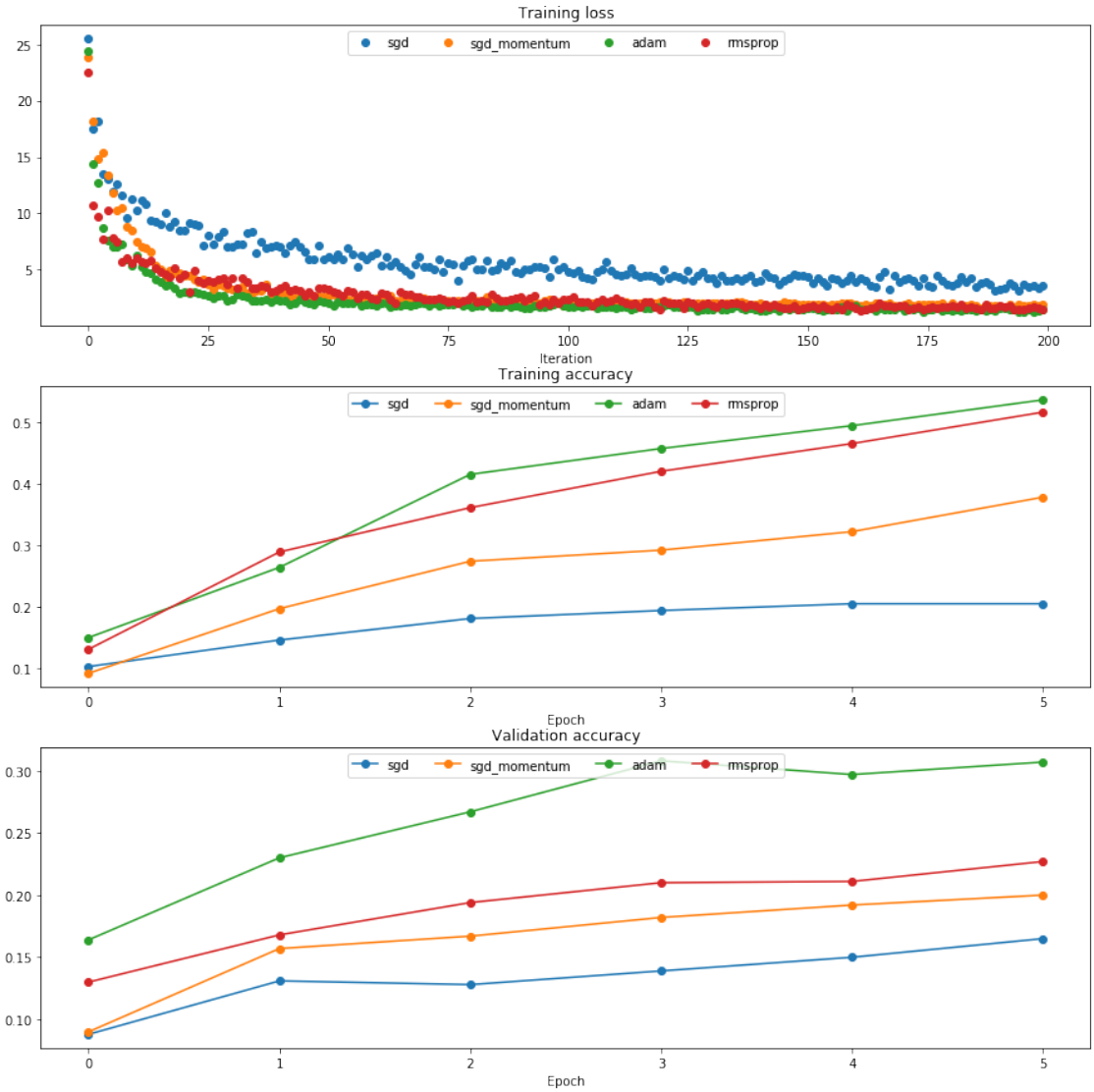
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```
In [128]: 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: 24.367495
(Epoch 0 / 5) train acc: 0.150000; val_acc: 0.164000
(Iteration 11 / 200) loss: 6.222972
(Iteration 21 / 200) loss: 2.998138
(Iteration 31 / 200) loss: 2.333535
(Epoch 1 / 5) train acc: 0.264000; val_acc: 0.230000
(Iteration 41 / 200) loss: 2.256579
(Iteration 51 / 200) loss: 2.053400
(Iteration 61 / 200) loss: 1.794105
(Iteration 71 / 200) loss: 2.074381
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.267000
```

```
(Iteration 81 / 200) loss: 1.659071
(Iteration 91 / 200) loss: 1.684889
(Iteration 101 / 200) loss: 1.673981
(Iteration 111 / 200) loss: 1.644885
(Epoch 3 / 5) train acc: 0.457000; val_acc: 0.308000
(Iteration 121 / 200) loss: 1.827878
(Iteration 131 / 200) loss: 1.517418
(Iteration 141 / 200) loss: 1.380265
(Iteration 151 / 200) loss: 1.576847
(Epoch 4 / 5) train acc: 0.494000; val_acc: 0.297000
(Iteration 161 / 200) loss: 1.793351
(Iteration 171 / 200) loss: 1.357383
(Iteration 181 / 200) loss: 1.407575
(Iteration 191 / 200) loss: 1.588814
(Epoch 5 / 5) train acc: 0.536000; val_acc: 0.307000

running with  rmsprop
(Iteration 1 / 200) loss: 22.469834
(Epoch 0 / 5) train acc: 0.131000; val_acc: 0.130000
(Iteration 11 / 200) loss: 6.077731
(Iteration 21 / 200) loss: 4.551940
(Iteration 31 / 200) loss: 4.246853
(Epoch 1 / 5) train acc: 0.289000; val_acc: 0.168000
(Iteration 41 / 200) loss: 3.365351
(Iteration 51 / 200) loss: 3.206939
(Iteration 61 / 200) loss: 2.367928
(Iteration 71 / 200) loss: 2.319942
(Epoch 2 / 5) train acc: 0.361000; val_acc: 0.194000
(Iteration 81 / 200) loss: 2.705558
(Iteration 91 / 200) loss: 2.645353
(Iteration 101 / 200) loss: 2.411135
(Iteration 111 / 200) loss: 2.467196
(Epoch 3 / 5) train acc: 0.420000; val_acc: 0.210000
(Iteration 121 / 200) loss: 2.233731
(Iteration 131 / 200) loss: 1.815985
(Iteration 141 / 200) loss: 1.587860
(Iteration 151 / 200) loss: 1.745673
(Epoch 4 / 5) train acc: 0.465000; val_acc: 0.211000
(Iteration 161 / 200) loss: 1.567292
(Iteration 171 / 200) loss: 1.809110
(Iteration 181 / 200) loss: 1.795550
(Iteration 191 / 200) loss: 1.860653
(Epoch 5 / 5) train acc: 0.516000; val_acc: 0.227000



/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:30:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:33:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
 ↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
 ↪this warning can be
```

```
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
↪ipykernel_launcher.py:36:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
↪ipykernel_launcher.py:40:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently␣
↪reuses the earlier
instance.  In a future version, a new instance will always be created and returned.  Meanwhile,␣
↪this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each axes instance.
```



13.1 Inline Question 3:

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

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

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

13.2 Answer:

Both Adagrad and Adam update the learning rate based on a per-weight basis, however Adagrad does so in such a way that the learning rate decays monotonically, causing learning to slow down, while Adam allows learning rates to increase along with weights.

14 Train a good model!

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

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

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

In [18]:
```
################################################################################
# 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.                                                     #
################################################################################

solver_data = {'X_train': data['X_train'],
               'y_train': data['y_train'],
               'X_val': data['X_val'],
               'y_val': data['y_val']
              }

def hyperparameter_search():
    learning_rates = np.arange(1e-3, 5e-3, 5e-4)
    reg_strengths = np.arange(1e-5, 1e-4, 5e-6)
    lr_decays = np.arange(0.85, 0.95, 0.01)
    decay_rates = np.arange(0.95, 0.99, 0.01)
    epsilons = np.arange(1e-8, 1e-7, 2e-9)
    weight_scales = np.arange(5e-4, 1e-1, 1e-4)

    best_acc = -np.Infinity
    best_model = None
    best_params = None

    while best_acc < 0.5:
        learning_rate = np.random.choice(learning_rates)
        reg_strength = np.random.choice(reg_strengths)
        lr_decay = np.random.choice(lr_decays)
```

```
        epsilon = np.random.choice(epsilons)
        decay_rate = np.random.choice(decay_rates)
        weight_scale = np.random.choice(weight_scales)

        optim_config = {'learning_rate': learning_rate,
                        'decay_rate': decay_rate,
                        'epsilon': epsilon}

        # preselected parameters based on manual tuning
        batch_size = 500
        hidden_dim = 100
        num_epochs = 20

        print('lr = %e, reg = %e, lr decay = %f, eps = %e, rms decay = %e' %
(learning_rate,
reg_strength,
                                                                    lr_decay,
                                                                    epsilon,
decay_rate))

        # model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)
        model = FullyConnectedNet([hidden_dim, hidden_dim, hidden_dim, hidden_dim,
hidden_dim], weight_scale=weight_scale)

        solver = Solver(model,
                        solver_data,
                        optim_config=optim_config,
                        update_rule='rmsprop',
                        lr_decay=lr_decay,
                        batch_size=batch_size,
                        num_epochs=num_epochs,
                        print_every=np.Infinity)

        solver.train()
        test_acc = solver.check_accuracy(data['X_test'], data['y_test'])
        print('test set accuracy = ', test_acc)

        if test_acc > best_acc:
            print('----- updating parameters -----')
            best_params = tuple([learning_rate, reg_strength, lr_decay])
            best_acc = test_acc
            best_model = model

    return best_model, best_params

# By default, the notebook will run the parameters found on a previous run of the
hyperparameter_search()
# function defined above, which resulted in a test accuracy of 55.8%. If you want to
perform hyperparameter
# search, change the value of the following variable to True
do_hyperparameter_search = False

if do_hyperparameter_search:
    model, params = hyperparameter_search()
else:
    # parameters found by running hyperparameter_search() resulting in a test accuracy
of 52.4%
    learning_rate = 3.3e-3
    reg_strength = 5.1e-5
    lr_decay = 0.87
    epsilon = 4e-8
    decay_rate = 0.96
    weight_scale = 5e-3

    optim_config = {'learning_rate': learning_rate,
                    'decay_rate': decay_rate,
                    'epsilon': epsilon}
```

```
    # preselected parameters based on manual tuning
    batch_size = 500
    hidden_dim = 100
    num_epochs = 20

    print('lr = %e, reg = %e, lr decay = %f, eps = %e, rms decay = %e' % (learning_rate,
                                                                          reg_strength,
                                                                          lr_decay,
                                                                          epsilon,
                                                                          decay_rate))

    model = FullyConnectedNet([hidden_dim, hidden_dim, hidden_dim, hidden_dim,
hidden_dim], weight_scale=weight_scale)

    solver = Solver(model,
                    solver_data,
                    optim_config=optim_config,
                    update_rule='rmsprop',
                    lr_decay=lr_decay,
                    batch_size=batch_size,
                    num_epochs=num_epochs,
                    print_every=np.Infinity)

    solver.train()
    test_acc = solver.check_accuracy(data['X_test'], data['y_test'])
    print('test set accuracy = ', test_acc)

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

```
lr = 3.300000e-03, reg = 5.100000e-05, lr decay = 0.870000, eps = 4.000000e-08, rms decay = 9.
↪600000e-01
(Iteration 1 / 1960) loss: 2.302585
(Epoch 0 / 20) train acc: 0.103000; val_acc: 0.087000
(Epoch 1 / 20) train acc: 0.253000; val_acc: 0.286000
(Epoch 2 / 20) train acc: 0.394000; val_acc: 0.395000
(Epoch 3 / 20) train acc: 0.384000; val_acc: 0.379000
(Epoch 4 / 20) train acc: 0.415000; val_acc: 0.419000
(Epoch 5 / 20) train acc: 0.476000; val_acc: 0.451000
(Epoch 6 / 20) train acc: 0.502000; val_acc: 0.495000
(Epoch 7 / 20) train acc: 0.540000; val_acc: 0.517000
(Epoch 8 / 20) train acc: 0.545000; val_acc: 0.508000
(Epoch 9 / 20) train acc: 0.569000; val_acc: 0.531000
(Epoch 10 / 20) train acc: 0.585000; val_acc: 0.512000
(Epoch 11 / 20) train acc: 0.577000; val_acc: 0.509000
(Epoch 12 / 20) train acc: 0.582000; val_acc: 0.523000
(Epoch 13 / 20) train acc: 0.610000; val_acc: 0.513000
(Epoch 14 / 20) train acc: 0.593000; val_acc: 0.508000
(Epoch 15 / 20) train acc: 0.617000; val_acc: 0.524000
(Epoch 16 / 20) train acc: 0.620000; val_acc: 0.525000
(Epoch 17 / 20) train acc: 0.640000; val_acc: 0.546000
(Epoch 18 / 20) train acc: 0.652000; val_acc: 0.542000
(Epoch 19 / 20) train acc: 0.650000; val_acc: 0.550000
(Epoch 20 / 20) train acc: 0.628000; val_acc: 0.531000
test set accuracy =  0.512
```

15 Test your model!

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

```
In [20]: y_test_pred = np.argmax(model.loss(data['X_test']), axis=1)
         y_val_pred = np.argmax(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:  0.55
Test set accuracy:  0.512
```

BatchNormalization

October 20, 2019

1 Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization which was proposed by [3] in 2015.

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [3] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, [3] proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

[3] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

```
In [1]: # As usual, a bit of setup
        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-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))))

        def print_mean_std(x,axis=0):
            print('  means: ', x.mean(axis=axis))
            print('  stds:  ', x.std(axis=axis))
            print()
```

```
In [2]: # Load the (preprocessed) CIFAR10 data.
        data = get_CIFAR10_data()
        for k, v in 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,)
```

1.1 Batch normalization: forward

In the file `cs682/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above would be helpful!

```
In [3]: # Check the training-time forward pass by checking means and variances
        # of features both before and after batch normalization

        # Simulate the forward pass for a two-layer network
        np.random.seed(231)
        N, D1, D2, D3 = 200, 50, 60, 3
        X = np.random.randn(N, D1)
        W1 = np.random.randn(D1, D2)
        W2 = np.random.randn(D2, D3)
        a = np.maximum(0, X.dot(W1)).dot(W2)

        print('Before batch normalization:')
        print_mean_std(a,axis=0)

        gamma = np.ones((D3,))
        beta = np.zeros((D3,))
        # Means should be close to zero and stds close to one
        print('After batch normalization (gamma=1, beta=0)')
        a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
        print_mean_std(a_norm,axis=0)

        gamma = np.asarray([1.0, 2.0, 3.0])
        beta = np.asarray([11.0, 12.0, 13.0])

        # Now means should be close to beta and stds close to gamma
        print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
        a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
        print_mean_std(a_norm,axis=0)
```

```
Before batch normalization:
  means:  [ -2.3814598  -13.18038246   1.91780462]
  stds:   [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means:  [5.32907052e-17 7.04991621e-17 1.85962357e-17]
  stds:   [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means:  [11. 12. 13.]
  stds:   [0.99999999 1.99999999 2.99999999]
```

```
In [4]: # Check the test-time forward pass by running the training-time
        # forward pass many times to warm up the running averages, and then
        # checking the means and variances of activations after a test-time
        # forward pass.

        np.random.seed(231)
        N, D1, D2, D3 = 200, 50, 60, 3
        W1 = np.random.randn(D1, D2)
        W2 = np.random.randn(D2, D3)

        bn_param = {'mode': 'train'}
        gamma = np.ones(D3)
        beta = np.zeros(D3)

        for t in range(50):
            X = np.random.randn(N, D1)
            a = np.maximum(0, X.dot(W1)).dot(W2)
            batchnorm_forward(a, gamma, beta, bn_param)

        bn_param['mode'] = 'test'
        X = np.random.randn(N, D1)
        a = np.maximum(0, X.dot(W1)).dot(W2)
        a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

        # Means should be close to zero and stds close to one, but will be
        # noisier than training-time forward passes.
        print('After batch normalization (test-time):')
        print_mean_std(a_norm,axis=0)
```

```
After batch normalization (test-time):
  means:  [-0.03927354 -0.04349152 -0.10452688]
  stds:   [1.01531428 1.01238373 0.97819988]
```

1.2 Batch normalization: backward

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Once you have finished, run the following to numerically check your backward pass.

```
In [5]: # Gradient check batchnorm backward pass
        np.random.seed(231)
        N, D = 4, 5
```

```
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

bn_param = {'mode': 'train'}
fx = lambda x: batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: batchnorm_forward(x, a, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, b, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
db_num = eval_numerical_gradient_array(fb, beta.copy(), dout)

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
#You should expect to see relative errors between 1e-13 and 1e-8
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.7029235612572515e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

1.3 Batch normalization: alternative backward

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can do a similar simplification for the batch normalization backward pass too.

Given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_N \end{bmatrix}$, we first calculate the mean $\mu = \frac{1}{N}\sum_{k=1}^{N} x_k$ and variance $v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2$.
With μ and v calculated, we can calculate the standard deviation $\sigma = \sqrt{v + \epsilon}$ and normalized data Y with $y_i = \frac{x_i - \mu}{\sigma}$.

The meat of our problem is to get $\frac{\partial L}{\partial X}$ from the upstream gradient $\frac{\partial L}{\partial Y}$. It might be challenging to directly reason about the gradients over X and Y - try reasoning about it in terms of x_i and y_i first.

You will need to come up with the derivations for $\frac{\partial L}{\partial x_i}$, by relying on the Chain Rule to first calculate the intermediate $\frac{\partial \mu}{\partial x_i}$, $\frac{\partial v}{\partial x_i}$, $\frac{\partial \sigma}{\partial x_i}$, then assemble these pieces to calculate $\frac{\partial y_i}{\partial x_i}$. You should make sure each of the intermediary steps are all as simple as possible.

After doing so, implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

```
In [83]: np.random.seed(231)
         N, D = 100, 500
```

```
        x = 5 * np.random.randn(N, D) + 12
        gamma = np.random.randn(D)
        beta = np.random.randn(D)
        dout = np.random.randn(N, D)

        bn_param = {'mode': 'train'}
        out, cache = batchnorm_forward(x, gamma, beta, bn_param)

        t1 = time.time()
        dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
        t2 = time.time()
        dx2, dgamma2, dbeta2 = batchnorm_backward_alt(dout, cache)
        t3 = time.time()

        print('dx difference: ', rel_error(dx1, dx2))
        print('dgamma difference: ', rel_error(dgamma1, dgamma2))
        print('dbeta difference: ', rel_error(dbeta1, dbeta2))
        print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
```

```
x norm shape:  (100, 500)
gamma shape: (500,)
sqrt var shape: (500,)
xbar shape: (500,)
x centered shape: (100, 500)
d beta shape: (500,)
d gamma shape: (500,)
dx difference:  1.1087665004214909e-12
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 4.08x
```

1.4 Fully Connected Nets with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs682/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

Concretely, when the `normalization` flag is set to `"batchnorm"` in the constructor, you should insert a batch normalization layer before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized. Once you are done, run the following to gradient-check your implementation.

HINT: You might find it useful to define an additional helper layer similar to those in the file `cs682/layer_utils.py`. If you decide to do so, do it in the file `cs682/classifiers/fc_net.py`.

```
In [6]: 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,))

        # You should expect losses between 1e-4~1e-10 for W,
        # losses between 1e-08~1e-10 for b,
        # and losses between 1e-08~1e-09 for beta and gammas.
        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,
                                    normalization='batchnorm')

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

```
          for name in sorted(grads):
            f = lambda _: model.loss(X, y)[0]
            grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
            print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
          if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 2.85e-06
W3 relative error: 4.05e-10
b1 relative error: 2.22e-07
b2 relative error: 2.22e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.33e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 6.96e-09
gamma2 relative error: 1.96e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.28e-06
W3 relative error: 1.11e-08
b1 relative error: 1.38e-08
b2 relative error: 8.33e-07
b3 relative error: 2.10e-10
beta1 relative error: 6.65e-09
beta2 relative error: 4.23e-09
gamma1 relative error: 6.27e-09
gamma2 relative error: 5.28e-09
```

2 Batchnorm for deep networks

Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

```
In [7]: np.random.seed(231)
        # Try training a very deep net with batchnorm
        hidden_dims = [100, 100, 100, 100, 100]

        num_train = 1000
        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 = 2e-2
        bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,
        normalization='batchnorm')
        model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=None)

        bn_solver = Solver(bn_model, small_data,
                        num_epochs=10, batch_size=50,
                        update_rule='adam',
                        optim_config={
                          'learning_rate': 1e-3,
                        },
```

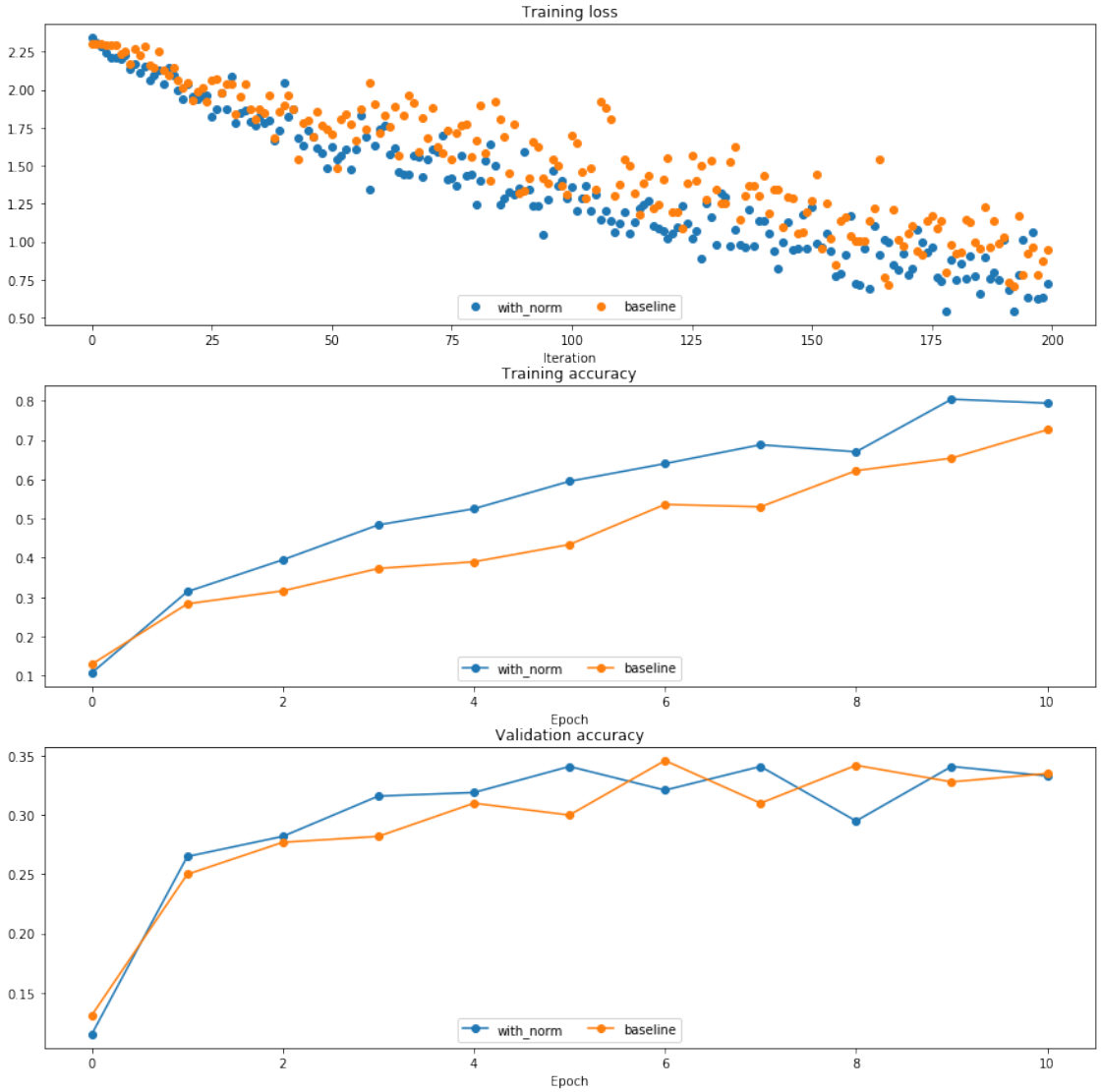
```
                    verbose=True,print_every=20)
bn_solver.train()

solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 200) loss: 2.340975
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.314000; val_acc: 0.265000
(Iteration 21 / 200) loss: 2.039345
(Epoch 2 / 10) train acc: 0.395000; val_acc: 0.282000
(Iteration 41 / 200) loss: 2.047471
(Epoch 3 / 10) train acc: 0.484000; val_acc: 0.316000
(Iteration 61 / 200) loss: 1.739554
(Epoch 4 / 10) train acc: 0.525000; val_acc: 0.319000
(Iteration 81 / 200) loss: 1.246973
(Epoch 5 / 10) train acc: 0.595000; val_acc: 0.341000
(Iteration 101 / 200) loss: 1.355908
(Epoch 6 / 10) train acc: 0.640000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.019017
(Epoch 7 / 10) train acc: 0.688000; val_acc: 0.341000
(Iteration 141 / 200) loss: 1.140477
(Epoch 8 / 10) train acc: 0.670000; val_acc: 0.295000
(Iteration 161 / 200) loss: 0.719171
(Epoch 9 / 10) train acc: 0.804000; val_acc: 0.341000
(Iteration 181 / 200) loss: 0.750465
(Epoch 10 / 10) train acc: 0.794000; val_acc: 0.333000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.131000
(Epoch 1 / 10) train acc: 0.283000; val_acc: 0.250000
(Iteration 21 / 200) loss: 2.041970
(Epoch 2 / 10) train acc: 0.316000; val_acc: 0.277000
(Iteration 41 / 200) loss: 1.900473
(Epoch 3 / 10) train acc: 0.373000; val_acc: 0.282000
(Iteration 61 / 200) loss: 1.713156
(Epoch 4 / 10) train acc: 0.390000; val_acc: 0.310000
(Iteration 81 / 200) loss: 1.662209
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696062
(Epoch 6 / 10) train acc: 0.536000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.550785
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.310000
(Iteration 141 / 200) loss: 1.436308
(Epoch 8 / 10) train acc: 0.622000; val_acc: 0.342000
(Iteration 161 / 200) loss: 1.000868
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.328000
(Iteration 181 / 200) loss: 0.925455
(Epoch 10 / 10) train acc: 0.726000; val_acc: 0.335000
```

Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

```
In [8]: def plot_training_history(title, label, baseline, bn_solvers, plot_fn, bl_marker='.',
        bn_marker='.', labels=None):
```

```
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.train_acc_history, bl_marker='-o', bn_marker='-o')
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.val_acc_history, bl_marker='-o', bn_marker='-o')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



3 Batch normalization and initialization

We will now run a small experiment to study the interaction of batch normalization and weight initialization.

The first cell will train 8-layer networks both with and without batch normalization using different scales for weight initialization. The second layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

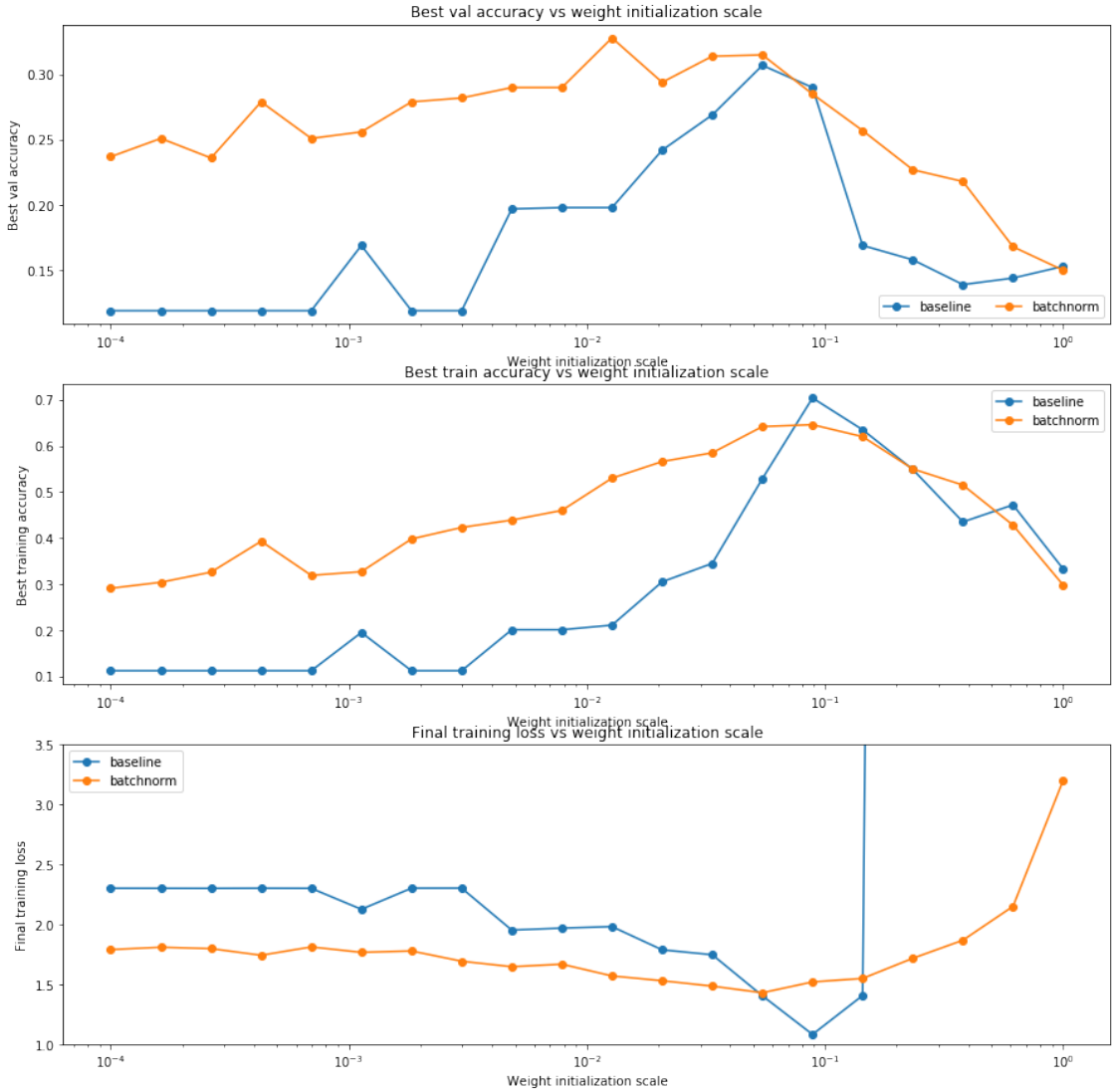
```
In [9]: np.random.seed(231)
        # Try training a very deep net with batchnorm
        hidden_dims = [50, 50, 50, 50, 50, 50, 50]
        num_train = 1000
        small_data = {
          'X_train': data['X_train'][:num_train],
```

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

bn_solvers_ws = {}
solvers_ws = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
  print('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
  bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,
normalization='batchnorm')
  model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=None)

  bn_solver = Solver(bn_model, small_data,
                     num_epochs=10, batch_size=50,
                     update_rule='adam',
                     optim_config={
                       'learning_rate': 1e-3,
                     },
                     verbose=False, print_every=200)
  bn_solver.train()
  bn_solvers_ws[weight_scale] = bn_solver

  solver = Solver(model, small_data,
                  num_epochs=10, batch_size=50,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 1e-3,
                  },
                  verbose=False, print_every=200)
  solver.train()
  solvers_ws[weight_scale] = solver
```

```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

```
In [10]: # Plot results of weight scale experiment
         best_train_accs, bn_best_train_accs = [], []
         best_val_accs, bn_best_val_accs = [], []
         final_train_loss, bn_final_train_loss = [], []

         for ws in weight_scales:
           best_train_accs.append(max(solvers_ws[ws].train_acc_history))
           bn_best_train_accs.append(max(bn_solvers_ws[ws].train_acc_history))
```

```
  best_val_accs.append(max(solvers_ws[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers_ws[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers_ws[ws].loss_history[-100:]))

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()
plt.gca().set_ylim(1.0, 3.5)

plt.gcf().set_size_inches(15, 15)
plt.show()
```



3.1 Inline Question 1:

Describe the results of this experiment. How does the scale of weight initialization affect models with/without batch normalization differently, and why?

3.2 Answer:

For smaller weight initializations, networks using batch normalization converge much faster than those without, which often don't converge at all until initialization weights are on the order of $\sim 10^{-2}$. However, as weight initialization increases to be on the order of $\sim 10^{-1}$, the networks begin to perform similarly in terms of training and validation accuracy.

The reason for the increased performance is because the activations that the network without batch normalization sees are effectively "too small," which can result in vanishing gradients and

dead ReLUs, whereas activations that the network with batch normalization sees are normalized and therefore are less likely to "kill" an ReLU compared to the un-batch-normalized network.

4 Batch normalization and batch size

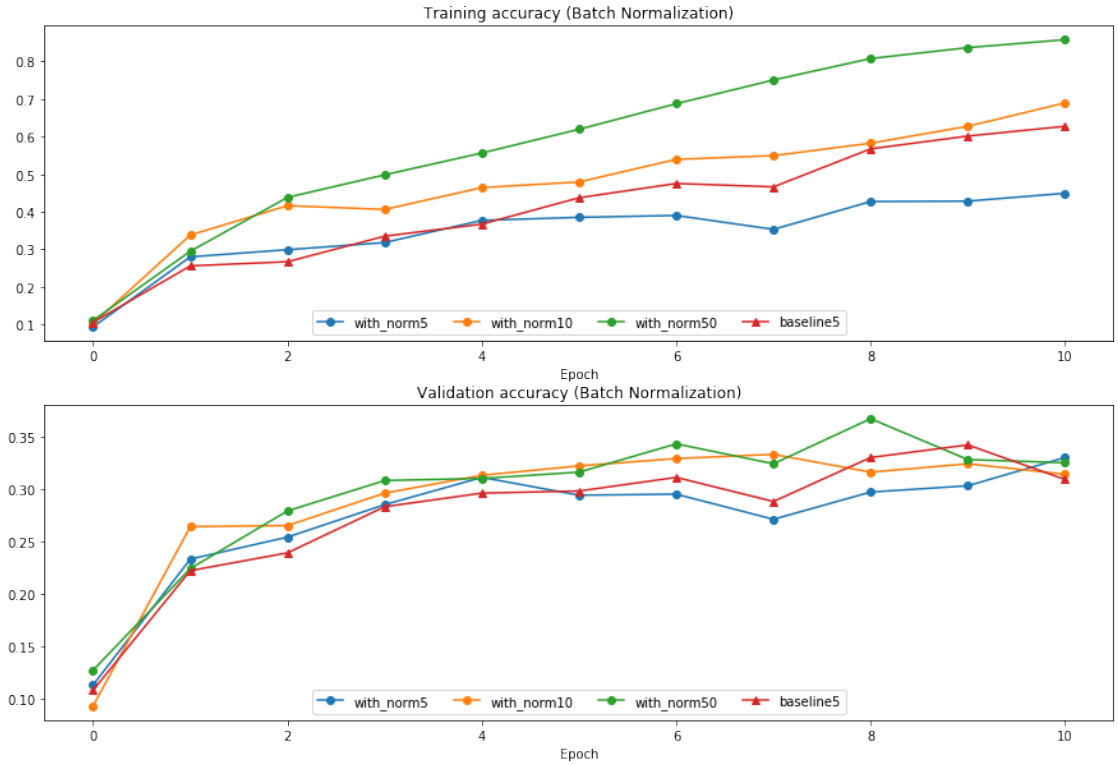
We will now run a small experiment to study the interaction of batch normalization and batch size.

The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

```
In [11]: def run_batchsize_experiments(normalization_mode):
             np.random.seed(231)
             # Try training a very deep net with batchnorm
             hidden_dims = [100, 100, 100, 100, 100]
             num_train = 1000
             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'],
             }
             n_epochs=10
             weight_scale = 2e-2
             batch_sizes = [5,10,50]
             lr = 10**(-3.5)
             solver_bsize = batch_sizes[0]

             print('No normalization: batch size = ',solver_bsize)
             model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,
         normalization=None)
             solver = Solver(model, small_data,
                             num_epochs=n_epochs, batch_size=solver_bsize,
                             update_rule='adam',
                             optim_config={
                               'learning_rate': lr,
                             },
                             verbose=False)
             solver.train()

             bn_solvers = []
             for i in range(len(batch_sizes)):
                 b_size=batch_sizes[i]
                 print('Normalization: batch size = ',b_size)
                 bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,
         normalization=normalization_mode)
                 bn_solver = Solver(bn_model, small_data,
                                    num_epochs=n_epochs, batch_size=b_size,
                                    update_rule='adam',
                                    optim_config={
                                      'learning_rate': lr,
                                    },
                                    verbose=False)
                 bn_solver.train()
                 bn_solvers.append(bn_solver)

             return bn_solvers, solver, batch_sizes

         batch_sizes = [5,10,50]
         bn_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('batchnorm')

No normalization: batch size =  5
Normalization: batch size =  5
```

```
Normalization: batch size =  10
Normalization: batch size =  50
```

```
In [95]: plt.subplot(2, 1, 1)
         plot_training_history('Training accuracy (Batch Normalization)','Epoch', solver_bsize,
         bn_solvers_bsize, \
                               lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o',
         labels=batch_sizes)
         plt.subplot(2, 1, 2)
         plot_training_history('Validation accuracy (Batch Normalization)','Epoch', solver_bsize,
         bn_solvers_bsize, \
                               lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o',
         labels=batch_sizes)

         plt.gcf().set_size_inches(15, 10)
         plt.show()
```



4.1 Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

4.2 Answer:

Batch normalization actually causes the network to perform worse when using a batch size of 5, while larger batch sizes result in increasing performance as we would expect.

This is because a batch size of 5 is almost too small, in the sense that the sample mean and variance are not good estimators of the true mean and variance of the data, and therefore batch normalization fails by possibly making more data very small (or less than zero), whereas batch sizes of 10 and 50 allow for more accurate approximation the true mean and variance of the data.

5 Layer Normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks which have a cap on the input batch size due to hardware limitations.

Several alternatives to batch normalization have been proposed to mitigate this problem; one such technique is Layer Normalization [4]. Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

[4] [Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.](https://arxiv.org/pdf/1607.06450.pdf)

5.1 Inline Question 3:

Which of these data preprocessing steps is analogous to batch normalization, and which is analogous to layer normalization?

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.

3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

5.2 Answer:

1. Layer normalization
2. Batch normalization
3. Batch normalization
4. Neither

6 Layer Normalization: Implementation

Now you'll implement layer normalization. This step should be relatively straightforward, as conceptually the implementation is almost identical to that of batch normalization. One significant difference though is that for layer normalization, we do not keep track of the moving moments, and the testing phase is identical to the training phase, where the mean and variance are directly calculated per datapoint.

Here's what you need to do:

- In `cs682/layers.py`, implement the forward pass for layer normalization in the function `layernorm_backward`.

Run the cell below to check your results. * In `cs682/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cs682/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNet`. When the `normalization` flag is set to `"layernorm"` in the constructor, you should insert a layer normalization layer before each ReLU nonlinearity.

Run the third cell below to run the batch size experiment on layer normalization.

In [16]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after layer normalization

# Simulate the forward pass for a two-layer network
np.random.seed(231)
N, D1, D2, D3 =4, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before layer normalization:')
print_mean_std(a,axis=1)

gamma = np.ones(D3)
beta = np.zeros(D3)
# Means should be close to zero and stds close to one
print('After layer normalization (gamma=1, beta=0)')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)

gamma = np.asarray([3.0,3.0,3.0])
beta = np.asarray([5.0,5.0,5.0])
# Now means should be close to beta and stds close to gamma
print('After layer normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)
```

```
Before layer normalization:
  means:  [-59.06673243 -47.60782686 -43.31137368 -26.40991744]
  stds:   [10.07429373 28.39478981 35.28360729  4.01831507]

After layer normalization (gamma=1, beta=0)
  means:  [ 4.81096644e-16 -7.40148683e-17  2.22044605e-16 -5.92118946e-16]
  stds:   [0.99999995 0.99999999 1.         0.99999969]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means:  [5. 5. 5. 5.]
  stds:   [2.99999985 2.99999998 2.99999999 2.99999907]
```

In [95]:
```
# Gradient check batchnorm backward pass
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

ln_param = {}
```

```
        fx = lambda x: layernorm_forward(x, gamma, beta, ln_param)[0]
        fg = lambda a: layernorm_forward(x, a, beta, ln_param)[0]
        fb = lambda b: layernorm_forward(x, gamma, b, ln_param)[0]

        dx_num = eval_numerical_gradient_array(fx, x, dout)
        da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
        db_num = eval_numerical_gradient_array(fb, beta.copy(), dout)

        _, cache = layernorm_forward(x, gamma, beta, ln_param)
        dx, dgamma, dbeta = layernorm_backward(dout, cache)

        #You should expect to see relative errors between 1e-12 and 1e-8
        print('dx error: ', rel_error(dx_num, dx))
        print('dgamma error: ', rel_error(da_num, dgamma))
        print('dbeta error: ', rel_error(db_num, dbeta))
```

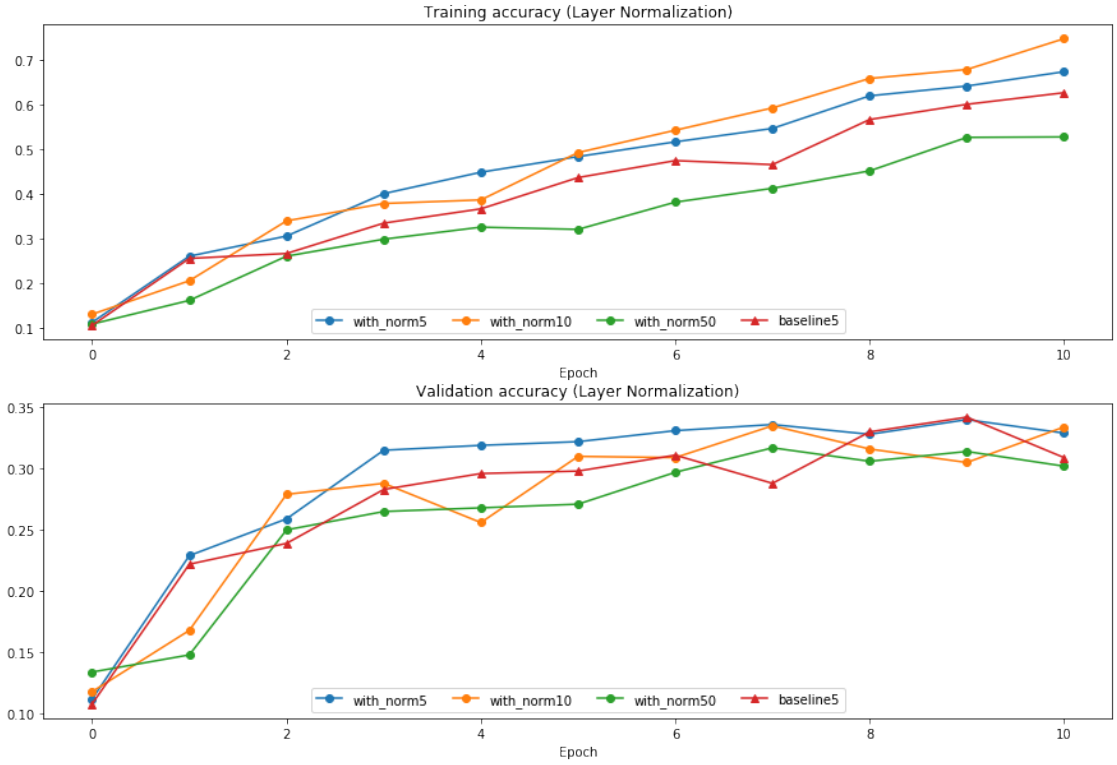
```
dx error:  1.4336161049967258e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

7 Layer Normalization and batch size

We will now run the previous batch size experiment with layer normalization instead of batch normalization. Compared to the previous experiment, you should see a markedly smaller influence of batch size on the training history!

```
In [98]: ln_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments('layernorm')

        plt.subplot(2, 1, 1)
        plot_training_history('Training accuracy (Layer Normalization)','Epoch', solver_bsize,
        ln_solvers_bsize, \
                              lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o',
        labels=batch_sizes)
        plt.subplot(2, 1, 2)
        plot_training_history('Validation accuracy (Layer Normalization)','Epoch', solver_bsize,
        ln_solvers_bsize, \
                              lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o',
        labels=batch_sizes)

        plt.gcf().set_size_inches(15, 10)
        plt.show()
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```



7.1 Inline Question 4:

When is layer normalization likely to not work well, and why?

1. Using it in a very deep network
2. Having a very small dimension of features
3. Having a high regularization term

7.2 Answer:

Dropout

October 20, 2019

1 Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] [Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012](https://arxiv.org/abs/1207.0580)

```
In [1]: # 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-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))))
```

```
In [2]: # Load the (preprocessed) CIFAR10 data.

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

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

2 Dropout forward pass

In the file `cs682/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

```
In [7]: np.random.seed(231)
        x = np.random.randn(500, 500) + 10

        for p in [0.25, 0.4, 0.7]:
            out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
            out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

            print('Running tests with p = ', p)
            print('Mean of input: ', x.mean())
            print('Mean of train-time output: ', out.mean())
            print('Mean of test-time output: ', out_test.mean())
            print('Fraction of train-time output set to zero: ', (out == 0).mean())
            print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
            print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

3 Dropout backward pass

In the file `cs682/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

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

        dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
        out, cache = dropout_forward(x, dropout_param)
        dx = dropout_backward(dout, cache)
        dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param)[0],
        x, dout)
        # print(dx_num)
```

```
        # Error should be around e-10 or less
        print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

3.1 Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2 Answer:

The outputs at test time will be smaller than they otherwise would be by a factor of p because we do not drop any neurons at test time. By multiplying by a factor of 1/p during training, we essentially cancel this scaling so the outputs have the same expected value.

4 Fully-connected nets with Dropout

In the file `cs682/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

```
In [9]: # 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 dropout in [1, 0.75, 0.5]:
          print('Running check with dropout = ', dropout)
          model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                                    weight_scale=5e-2, dtype=np.float64,
                                    dropout=dropout, seed=123)

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

          # Relative errors should be around e-6 or less; Note that it's fine
          # if for dropout=1 you have W2 error be on the order of e-5.
          for name in sorted(grads):
            f = lambda _: model.loss(X, y)[0]
            grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
            print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
          print()
```

```
Running check with dropout =  1
Initial loss:  2.3051948273987857
W1 relative error: 5.25e-07
W2 relative error: 1.89e-05
W3 relative error: 2.92e-07
b1 relative error: 1.34e-07
b2 relative error: 7.09e-08
b3 relative error: 1.49e-10

Running check with dropout =  0.75
```

```
Initial loss:  2.29898614757146
W1 relative error: 9.74e-07
W2 relative error: 4.12e-08
W3 relative error: 2.89e-08
b1 relative error: 9.62e-09
b2 relative error: 1.90e-09
b3 relative error: 8.93e-11

Running check with dropout =  0.5
Initial loss:  2.302437587710995
W1 relative error: 4.55e-08
W2 relative error: 2.97e-08
W3 relative error: 5.99e-07
b1 relative error: 1.87e-08
b2 relative error: 1.38e-09
b3 relative error: 1.84e-10
```

5 Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

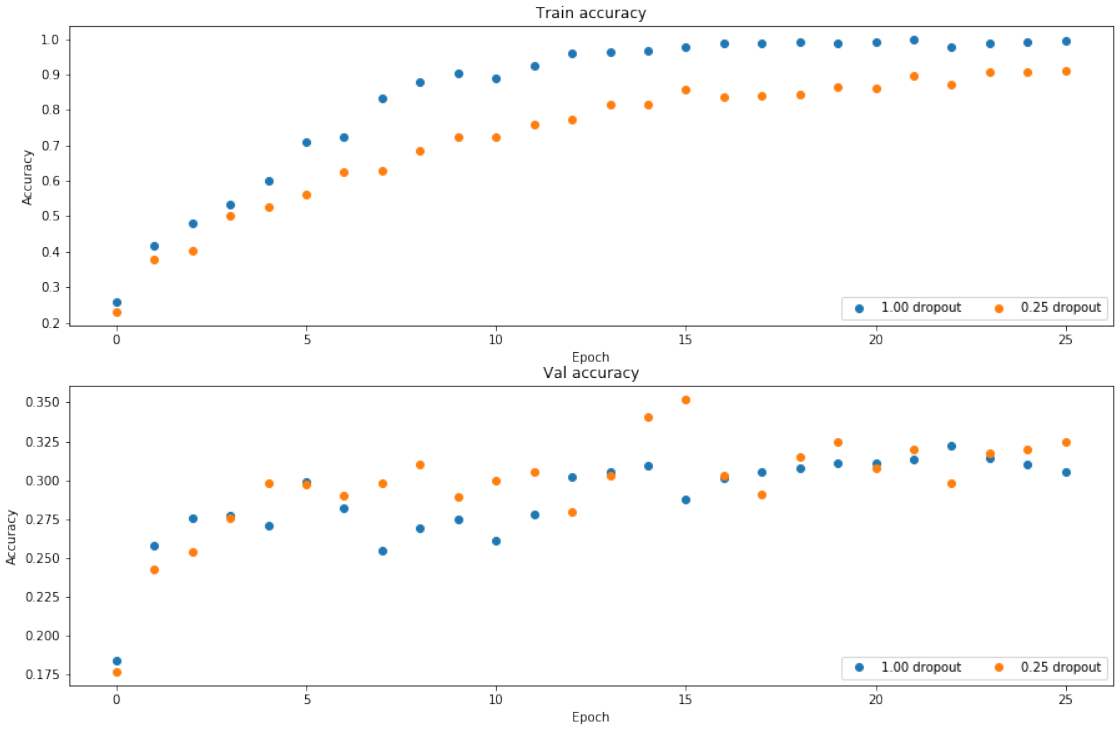
```
In [10]: # Train two identical nets, one with dropout and one without
         np.random.seed(231)
         num_train = 500
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': data['y_val'],
         }

         solvers = {}
         dropout_choices = [1, 0.25]
         for dropout in dropout_choices:
             model = FullyConnectedNet([500], dropout=dropout)
             print(dropout)

             solver = Solver(model,
                             small_data,
                             num_epochs=25,
                             batch_size=100,
                             update_rule='adam',
                             optim_config={'learning_rate': 5e-4,
                                           },
                             verbose=True,
                             print_every=100)
             solver.train()
             solvers[dropout] = solver
```

```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
```

```
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.878000; val_acc: 0.269000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.275000
(Epoch 10 / 25) train acc: 0.888000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.926000; val_acc: 0.278000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.302000
(Epoch 13 / 25) train acc: 0.964000; val_acc: 0.305000
(Epoch 14 / 25) train acc: 0.966000; val_acc: 0.309000
(Epoch 15 / 25) train acc: 0.976000; val_acc: 0.288000
(Epoch 16 / 25) train acc: 0.988000; val_acc: 0.301000
(Epoch 17 / 25) train acc: 0.988000; val_acc: 0.305000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.308000
(Epoch 19 / 25) train acc: 0.988000; val_acc: 0.311000
(Epoch 20 / 25) train acc: 0.990000; val_acc: 0.311000
(Iteration 101 / 125) loss: 0.006070
(Epoch 21 / 25) train acc: 0.998000; val_acc: 0.313000
(Epoch 22 / 25) train acc: 0.976000; val_acc: 0.322000
(Epoch 23 / 25) train acc: 0.986000; val_acc: 0.314000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.310000
(Epoch 25 / 25) train acc: 0.994000; val_acc: 0.305000
0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.297000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.290000
(Epoch 7 / 25) train acc: 0.628000; val_acc: 0.298000
(Epoch 8 / 25) train acc: 0.686000; val_acc: 0.310000
(Epoch 9 / 25) train acc: 0.722000; val_acc: 0.289000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.300000
(Epoch 11 / 25) train acc: 0.760000; val_acc: 0.305000
(Epoch 12 / 25) train acc: 0.772000; val_acc: 0.280000
(Epoch 13 / 25) train acc: 0.814000; val_acc: 0.303000
(Epoch 14 / 25) train acc: 0.814000; val_acc: 0.341000
(Epoch 15 / 25) train acc: 0.856000; val_acc: 0.352000
(Epoch 16 / 25) train acc: 0.838000; val_acc: 0.303000
(Epoch 17 / 25) train acc: 0.840000; val_acc: 0.291000
(Epoch 18 / 25) train acc: 0.844000; val_acc: 0.315000
(Epoch 19 / 25) train acc: 0.864000; val_acc: 0.325000
(Epoch 20 / 25) train acc: 0.862000; val_acc: 0.308000
(Iteration 101 / 125) loss: 5.259476
(Epoch 21 / 25) train acc: 0.896000; val_acc: 0.320000
(Epoch 22 / 25) train acc: 0.872000; val_acc: 0.298000
(Epoch 23 / 25) train acc: 0.906000; val_acc: 0.317000
(Epoch 24 / 25) train acc: 0.906000; val_acc: 0.320000
(Epoch 25 / 25) train acc: 0.910000; val_acc: 0.325000
```

In [11]:
```
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])
```

```
plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



5.1 Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

5.2 Answer:

The network using dropout has a lower training accuracy than the one without, while they have similar validation accuracies, suggesting that dropout is an effective regularizer because it prevents the network from over-fitting on the training data.

5.3 Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

5.4 Answer:

We should not modify p.

ConvolutionalNetworks

October 20, 2019

1 Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

```
In [1]: # As usual, a bit of setup
        import numpy as np
        import matplotlib.pyplot as plt
        from cs682.classifiers.cnn import *
        from cs682.data_utils import get_CIFAR10_data
        from cs682.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
        from cs682.layers import *
        from cs682.fast_layers import *
        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-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))))
```

```
In [2]: # Load the (preprocessed) CIFAR10 data.

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

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

2 Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs682/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

```
In [3]: x_shape = (2, 3, 4, 4)
        w_shape = (3, 3, 4, 4)
        x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
        w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
        b = np.linspace(-0.1, 0.2, num=3)

        conv_param = {'stride': 2, 'pad': 1}
        out, _ = conv_forward_naive(x, w, b, conv_param)
        correct_out = np.array([[[[-0.08759809, -0.10987781],
                                  [-0.18387192, -0.2109216 ]],
                                 [[ 0.21027089,  0.21661097],
                                  [ 0.22847626,  0.23004637]],
                                 [[ 0.50813986,  0.54309974],
                                  [ 0.64082444,  0.67101435]]],
                                [[[-0.98053589, -1.03143541],
                                  [-1.19128892, -1.24695841]],
                                 [[ 0.69108355,  0.66880383],
                                  [ 0.59480972,  0.56776003]],
                                 [[ 2.36270298,  2.36904306],
                                  [ 2.38090835,  2.38247847]]]])

        # Compare your output to ours; difference should be around e-8
        print('Testing conv_forward_naive')
        print('difference: ', rel_error(out, correct_out))

Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

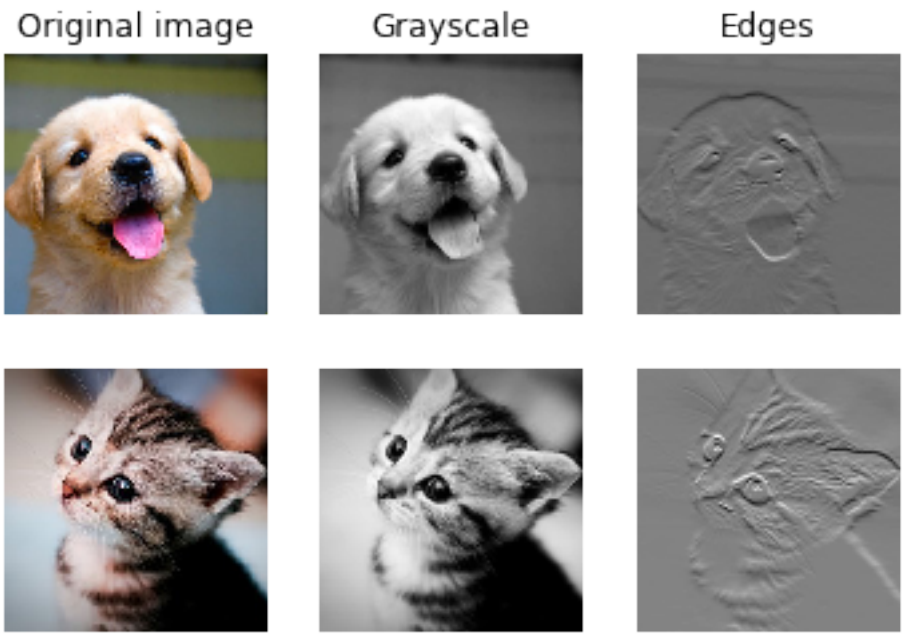
3 Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

```
In [9]: from scipy.misc import imread, imresize

        kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
        # kitten is wide, and puppy is already square
        d = kitten.shape[1] - kitten.shape[0]
        kitten_cropped = kitten[:, d//2:-d//2, :]

        img_size = 200   # Make this smaller if it runs too slow
        x = np.zeros((2, 3, img_size, img_size))
        x[0, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
        x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size)).transpose((2, 0, 1))

        # Set up a convolutional weights holding 2 filters, each 3x3
        w = np.zeros((2, 3, 3, 3))
```

```
        # The first filter converts the image to grayscale.
        # Set up the red, green, and blue channels of the filter.
        w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
        w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
        w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

        # Second filter detects horizontal edges in the blue channel.
        w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

        # Vector of biases. We don't need any bias for the grayscale
        # filter, but for the edge detection filter we want to add 128
        # to each output so that nothing is negative.
        b = np.array([0, 128])

        # Compute the result of convolving each input in x with each filter in w,
        # offsetting by b, and storing the results in out.
        out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

        def imshow_noax(img, normalize=True):
            """ Tiny helper to show images as uint8 and remove axis labels """
            if normalize:
                img_max, img_min = np.max(img), np.min(img)
                img = 255.0 * (img - img_min) / (img_max - img_min)
            plt.imshow(img.astype('uint8'))
            plt.gca().axis('off')

        # Show the original images and the results of the conv operation
        plt.subplot(2, 3, 1)
        imshow_noax(puppy, normalize=False)
        plt.title('Original image')
        plt.subplot(2, 3, 2)
        imshow_noax(out[0, 0])
        plt.title('Grayscale')
        plt.subplot(2, 3, 3)
        imshow_noax(out[0, 1])
        plt.title('Edges')
        plt.subplot(2, 3, 4)
        imshow_noax(kitten_cropped, normalize=False)
        plt.subplot(2, 3, 5)
        imshow_noax(out[1, 0])
        plt.subplot(2, 3, 6)
        imshow_noax(out[1, 1])
        plt.show()
```

```
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:3:
DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:10:
DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
/home/brendan/Desktop/school/grad/f19/cs682/assignment2/env/lib/python3.6/site-packages/
 ↪ipykernel_launcher.py:11:
DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



4 Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs682/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

```
In [4]: np.random.seed(231)
        x = np.random.randn(4, 3, 5, 5)
        w = np.random.randn(2, 3, 3, 3)
        b = np.random.randn(2,)
        dout = np.random.randn(4, 2, 5, 5)
        conv_param = {'stride': 1, 'pad': 1}
        # print('actual dout shape: ', dout.shape)

        dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b,
        conv_param)[0], x, dout)
        dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,
        conv_param)[0], w, dout)
        db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,
        conv_param)[0], b, dout)
        # print('dx num shape: ', dx_num.shape)
        # print()
        # print(dx_num)

        out, cache = conv_forward_naive(x, w, b, conv_param)
        dx, dw, db = conv_backward_naive(dout, cache)
        # print('final dx shape: ', dx.shape)

        # Your errors should be around e-8 or less.
        print('Testing conv_backward_naive function')
        print('dx error: ', rel_error(dx, dx_num))
```

```
        print('dw error: ', rel_error(dw, dw_num))
        print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

5 Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs682/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
In [5]: x_shape = (2, 3, 4, 4)
        x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
        pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

        out, _ = max_pool_forward_naive(x, pool_param)

        correct_out = np.array([[[[-0.26315789, -0.24842105],
                                  [-0.20421053, -0.18947368]],
                                 [[-0.14526316, -0.13052632],
                                  [-0.08631579, -0.07157895]],
                                 [[-0.02736842, -0.01263158],
                                  [ 0.03157895,  0.04631579]]],
                                [[[ 0.09052632,  0.10526316],
                                  [ 0.14947368,  0.16421053]],
                                 [[ 0.20842105,  0.22315789],
                                  [ 0.26736842,  0.28210526]],
                                 [[ 0.32631579,  0.34105263],
                                  [ 0.38526316,  0.4       ]]]])

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

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

6 Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs682/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

```
In [6]: np.random.seed(231)
        x = np.random.randn(3, 2, 8, 8)
        dout = np.random.randn(3, 2, 4, 4)
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

```
        out, cache = max_pool_forward_naive(x, pool_param)
        dx = max_pool_backward_naive(dout, cache)

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

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

7 Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs682/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs682` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

```
In [13]: # Rel errors should be around e-9 or less
         from cs682.fast_layers import conv_forward_fast, conv_backward_fast
         from time import time
         np.random.seed(231)
         x = np.random.randn(100, 3, 31, 31)
         w = np.random.randn(25, 3, 3, 3)
         b = np.random.randn(25,)
         dout = np.random.randn(100, 25, 16, 16)
         conv_param = {'stride': 2, 'pad': 1}

         t0 = time()
         out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
         t1 = time()
         out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
         t2 = time()

         print('Testing conv_forward_fast:')
         print('Naive: %fs' % (t1 - t0))
         print('Fast: %fs' % (t2 - t1))
         print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
         print('Difference: ', rel_error(out_naive, out_fast))

         t0 = time()
         dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
         t1 = time()
         dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
         t2 = time()
```

```
print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 5.908555s
Fast: 0.010827s
Speedup: 545.744858x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 8.295317s
Fast: 0.011106s
Speedup: 746.953199x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  0.0
```

In [14]:
```
# Relative errors should be close to 0.0
from cs682.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.482915s
fast: 0.002104s
speedup: 229.517507x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.702751s
fast: 0.014032s
```

```
speedup: 121.350721x
dx difference:  0.0
```

8 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs682/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

```
In [15]: from cs682.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
         np.random.seed(231)
         x = np.random.randn(2, 3, 16, 16)
         w = np.random.randn(3, 3, 3, 3)
         b = np.random.randn(3,)
         dout = np.random.randn(2, 3, 8, 8)
         conv_param = {'stride': 1, 'pad': 1}
         pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

         out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
         dx, dw, db = conv_relu_pool_backward(dout, cache)

         dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b,
         conv_param, pool_param)[0], x, dout)
         dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b,
         conv_param, pool_param)[0], w, dout)
         db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b,
         conv_param, pool_param)[0], b, dout)

         # Relative errors should be around e-8 or less
         print('Testing conv_relu_pool')
         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 conv_relu_pool
dx error:  9.591132621921372e-09
dw error:  5.802391137330214e-09
db error:  1.0146343411762047e-09
```

```
In [16]: from cs682.layer_utils import conv_relu_forward, conv_relu_backward
         np.random.seed(231)
         x = np.random.randn(2, 3, 8, 8)
         w = np.random.randn(3, 3, 3, 3)
         b = np.random.randn(3,)
         dout = np.random.randn(2, 3, 8, 8)
         conv_param = {'stride': 1, 'pad': 1}

         out, cache = conv_relu_forward(x, w, b, conv_param)
         dx, dw, db = conv_relu_backward(dout, cache)

         dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b,
         conv_param)[0], x, dout)
         dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,
         conv_param)[0], w, dout)
         db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,
         conv_param)[0], b, dout)

         # Relative errors should be around e-8 or less
         print('Testing conv_relu:')
         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 conv_relu:
dx error:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

9 Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs682/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

9.1 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

```
In [36]: model = ThreeLayerConvNet()

         N = 50
         X = np.random.randn(N, 3, 32, 32)
         y = np.random.randint(10, size=N)

         loss, grads = model.loss(X, y)
         print('Initial loss (no regularization): ', loss)

         model.reg = 0.5
         loss, grads = model.loss(X, y)
         print('Initial loss (with regularization): ', loss)
```

```
filter dim:  (32, 3, 7, 7)
pool dim:  (8192, 100)
Initial loss (no regularization):  2.3025866831027293
Initial loss (with regularization):  2.50808657056916
```

9.2 Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

```
In [37]: num_inputs = 2
         input_dim = (3, 16, 16)
         reg = 0.0
         num_classes = 10
         np.random.seed(231)
         X = np.random.randn(num_inputs, *input_dim)
         y = np.random.randint(num_classes, size=num_inputs)

         model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                                   input_dim=input_dim, hidden_dim=7,
```

```
                                dtype=np.float64)
        loss, grads = model.loss(X, y)
        # Errors should be small, but correct implementations may have
        # relative errors up to the order of e-2
        for param_name in sorted(grads):
            f = lambda _: model.loss(X, y)[0]
            param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=False,
        h=1e-6)
            e = rel_error(param_grad_num, grads[param_name])
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,
        grads[param_name])))
```

```
W1 max relative error: 1.380104e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.064049e-04
b1 max relative error: 3.477652e-05
b2 max relative error: 2.516375e-03
b3 max relative error: 7.945660e-10
```

9.3 Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

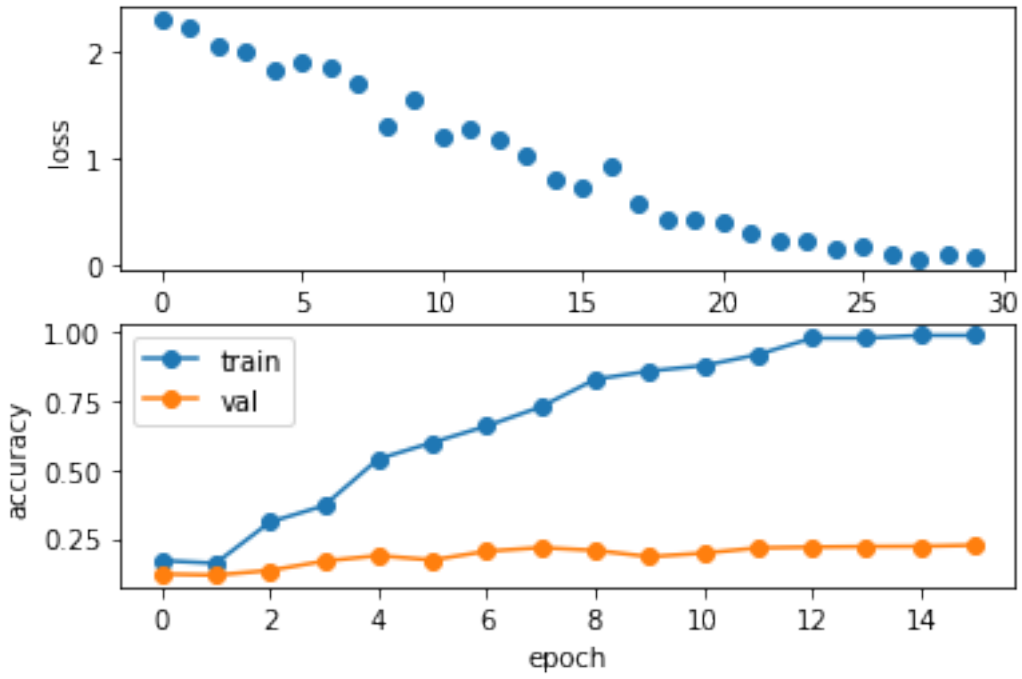
In [40]:
```
np.random.seed(231)

num_train = 100
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'],
}

model = ThreeLayerConvNet(weight_scale=5e-3)

solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 5e-4,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.307471
(Epoch 0 / 15) train acc: 0.170000; val_acc: 0.121000
(Iteration 2 / 30) loss: 2.218503
(Epoch 1 / 15) train acc: 0.160000; val_acc: 0.117000
(Iteration 3 / 30) loss: 2.057796
(Iteration 4 / 30) loss: 2.008379
(Epoch 2 / 15) train acc: 0.310000; val_acc: 0.134000
(Iteration 5 / 30) loss: 1.827644
(Iteration 6 / 30) loss: 1.895794
(Epoch 3 / 15) train acc: 0.370000; val_acc: 0.168000
(Iteration 7 / 30) loss: 1.846482
(Iteration 8 / 30) loss: 1.711759
(Epoch 4 / 15) train acc: 0.540000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.316028
(Iteration 10 / 30) loss: 1.561760
```

```
(Epoch 5 / 15) train acc: 0.600000; val_acc: 0.172000
(Iteration 11 / 30) loss: 1.214942
(Iteration 12 / 30) loss: 1.289294
(Epoch 6 / 15) train acc: 0.660000; val_acc: 0.204000
(Iteration 13 / 30) loss: 1.181641
(Iteration 14 / 30) loss: 1.037325
(Epoch 7 / 15) train acc: 0.730000; val_acc: 0.217000
(Iteration 15 / 30) loss: 0.802920
(Iteration 16 / 30) loss: 0.741001
(Epoch 8 / 15) train acc: 0.830000; val_acc: 0.207000
(Iteration 17 / 30) loss: 0.929612
(Iteration 18 / 30) loss: 0.589986
(Epoch 9 / 15) train acc: 0.860000; val_acc: 0.184000
(Iteration 19 / 30) loss: 0.427690
(Iteration 20 / 30) loss: 0.433301
(Epoch 10 / 15) train acc: 0.880000; val_acc: 0.197000
(Iteration 21 / 30) loss: 0.404702
(Iteration 22 / 30) loss: 0.309422
(Epoch 11 / 15) train acc: 0.920000; val_acc: 0.216000
(Iteration 23 / 30) loss: 0.229483
(Iteration 24 / 30) loss: 0.237907
(Epoch 12 / 15) train acc: 0.980000; val_acc: 0.219000
(Iteration 25 / 30) loss: 0.153805
(Iteration 26 / 30) loss: 0.178712
(Epoch 13 / 15) train acc: 0.980000; val_acc: 0.221000
(Iteration 27 / 30) loss: 0.113762
(Iteration 28 / 30) loss: 0.064097
(Epoch 14 / 15) train acc: 0.990000; val_acc: 0.222000
(Iteration 29 / 30) loss: 0.101137
(Iteration 30 / 30) loss: 0.089280
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.226000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

```
In [41]: plt.subplot(2, 1, 1)
         plt.plot(solver.loss_history, 'o')
         plt.xlabel('iteration')
         plt.ylabel('loss')

         plt.subplot(2, 1, 2)
         plt.plot(solver.train_acc_history, '-o')
         plt.plot(solver.val_acc_history, '-o')
         plt.legend(['train', 'val'], loc='upper left')
         plt.xlabel('epoch')
         plt.ylabel('accuracy')
         plt.show()
```



9.4 Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

```
In [42]: model = ThreeLayerConvNet(weight_scale=5e-4, hidden_dim=500, reg=1e-3)

         solver = Solver(model, data,
                         num_epochs=1, batch_size=50,
                         update_rule='adam',
                         optim_config={
                             'learning_rate': 5e-4
                         },
                         verbose=True, print_every=20)
         solver.train()
```

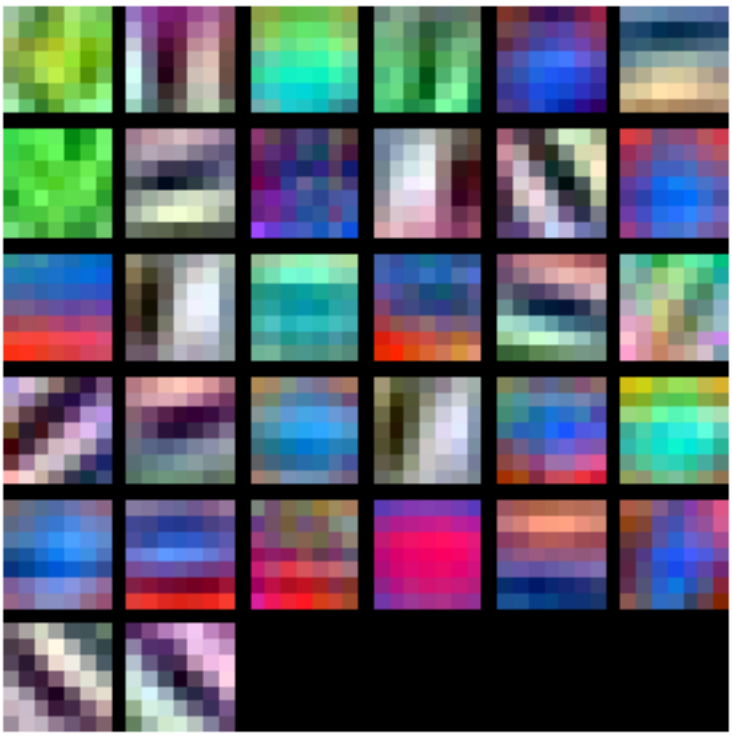
```
(Iteration 1 / 980) loss: 2.303111
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.183273
(Iteration 41 / 980) loss: 1.875637
(Iteration 61 / 980) loss: 1.858607
(Iteration 81 / 980) loss: 1.697790
(Iteration 101 / 980) loss: 1.764538
(Iteration 121 / 980) loss: 1.852278
(Iteration 141 / 980) loss: 1.776354
(Iteration 161 / 980) loss: 1.618060
(Iteration 181 / 980) loss: 1.692469
(Iteration 201 / 980) loss: 1.868310
(Iteration 221 / 980) loss: 1.760287
(Iteration 241 / 980) loss: 1.667674
(Iteration 261 / 980) loss: 1.570460
```

```
(Iteration 281 / 980) loss: 1.622974
(Iteration 301 / 980) loss: 1.579306
(Iteration 321 / 980) loss: 1.695323
(Iteration 341 / 980) loss: 1.435818
(Iteration 361 / 980) loss: 1.679720
(Iteration 381 / 980) loss: 1.311219
(Iteration 401 / 980) loss: 1.617242
(Iteration 421 / 980) loss: 1.296234
(Iteration 441 / 980) loss: 1.688545
(Iteration 461 / 980) loss: 1.732104
(Iteration 481 / 980) loss: 1.435397
(Iteration 501 / 980) loss: 1.278202
(Iteration 521 / 980) loss: 1.700642
(Iteration 541 / 980) loss: 1.581997
(Iteration 561 / 980) loss: 1.747323
(Iteration 581 / 980) loss: 1.217915
(Iteration 601 / 980) loss: 1.302511
(Iteration 621 / 980) loss: 1.396910
(Iteration 641 / 980) loss: 1.434761
(Iteration 661 / 980) loss: 1.442672
(Iteration 681 / 980) loss: 1.838162
(Iteration 701 / 980) loss: 1.417289
(Iteration 721 / 980) loss: 1.491407
(Iteration 741 / 980) loss: 1.529125
(Iteration 761 / 980) loss: 1.466295
(Iteration 781 / 980) loss: 1.743868
(Iteration 801 / 980) loss: 1.615121
(Iteration 821 / 980) loss: 1.265205
(Iteration 841 / 980) loss: 1.400344
(Iteration 861 / 980) loss: 1.454933
(Iteration 881 / 980) loss: 1.407469
(Iteration 901 / 980) loss: 1.253930
(Iteration 921 / 980) loss: 1.414958
(Iteration 941 / 980) loss: 1.466384
(Iteration 961 / 980) loss: 1.330837
(Epoch 1 / 1) train acc: 0.555000; val_acc: 0.553000
```

9.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

```
In [43]: from cs682.vis_utils import visualize_grid

         grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
         plt.imshow(grid.astype('uint8'))
         plt.axis('off')
         plt.gcf().set_size_inches(5, 5)
         plt.show()
```



10 Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different imagesand different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

[3] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

10.1 Spatial batch normalization: forward

In the file `cs682/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

In [7]:
```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 6.18949336e-16  5.99520433e-16 -1.22124533e-16]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

In [8]:
```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

10.2 Spatial batch normalization: backward

In the file `cs682/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

```
In [9]: np.random.seed(231)
        N, C, H, W = 2, 3, 4, 5
        x = 5 * np.random.randn(N, C, H, W) + 12
        gamma = np.random.randn(C)
        beta = np.random.randn(C)
        dout = np.random.randn(N, C, H, W)

        bn_param = {'mode': 'train'}
        fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
        fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
        fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

        dx_num = eval_numerical_gradient_array(fx, x, dout)
        da_num = eval_numerical_gradient_array(fg, gamma, dout)
        db_num = eval_numerical_gradient_array(fb, beta, dout)

        #You should expect errors of magnitudes between 1e-12~1e-06
        _, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
        dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
        print('dx error: ', rel_error(dx_num, dx))
        print('dgamma error: ', rel_error(da_num, dgamma))
        print('dbeta error: ', rel_error(db_num, dbeta))
```

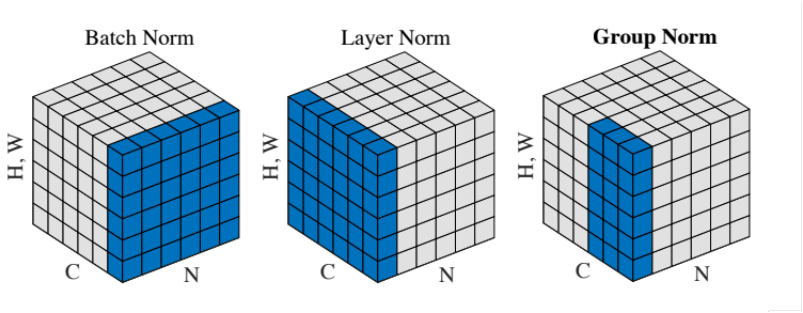
```
dx error:  2.786648193872555e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

11 Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.



Visual comparison of the normalization techniques discussed so far (image edited from [5])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]-- after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* -- this truly is still an ongoing and excitingly active field of research!

[4] [Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.](https://arxiv.org/pdf/1607.06450.pdf)

[5] [Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).](https://arxiv.org/abs/1803.08494)

[6] [N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.](https://ieeexplore.ieee.org/abstract/document/1467360/)

11.1 Group normalization: forward

In the file `cs682/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

```
In [53]: np.random.seed(231)
         # Check the training-time forward pass by checking means and variances
         # of features both before and after spatial batch normalization

         N, C, H, W = 2, 6, 4, 5
         G = 2
         x = 4 * np.random.randn(N, C, H, W) + 10
         x_g = x.reshape((N*G,-1))
         print('Before spatial group normalization:')
         print('  Shape: ', x.shape)
         print('  Means: ', x_g.mean(axis=1))
         print('  Stds: ', x_g.std(axis=1))

         # Means should be close to zero and stds close to one
         gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
         bn_param = {'mode': 'train'}

         out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
         out_g = out.reshape((N*G,-1))
         print('After spatial group normalization:')
         print('  Shape: ', out.shape)
```

```
        print('  Means: ', out_g.mean(axis=1))
        print('  Stds: ', out_g.std(axis=1))

Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
reshaped x:  (4, 60)
mean shape:  (60,)  var shape:  (60,)
x norm shape:  (4, 60)
After spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [2.83916409e-17 3.88578059e-17 2.03540888e-16 1.12872674e-16]
  Stds:  [0.9999987  0.99999895 0.9999988  0.99999936]
```

11.2 Spatial group normalization: backward

In the file `cs682/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

```
In [85]: np.random.seed(231)
         N, C, H, W = 2, 6, 4, 5
         G = 2
         x = 5 * np.random.randn(N, C, H, W) + 12
         gamma = np.random.randn(1,C,1,1)
         beta = np.random.randn(1,C,1,1)
         dout = np.random.randn(N, C, H, W)

         gn_param = {}
         fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
         fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
         fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

         dx_num = eval_numerical_gradient_array(fx, x, dout)
         da_num = eval_numerical_gradient_array(fg, gamma, dout)
         db_num = eval_numerical_gradient_array(fb, beta, dout)
         # print(dx_num)

         _, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
         dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
         # print(dx)
         #You should expect errors of magnitudes between 1e-12~1e-07
         print('dx error: ', rel_error(dx_num, dx))
         print('dgamma error: ', rel_error(da_num, dgamma))
         print('dbeta error: ', rel_error(db_num, dbeta))

x norm init shape:  (2, 6, 4, 5)
dx error:  7.413109437563619e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```