FullyConnectedNets

May 4, 2018

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

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

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

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

```
In [3]: # Load the (preprocessed) CIFAR10 data.

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

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

2 Affine layer: foward

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

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

```
In [33]: # Test the affine_forward function

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

         input_size = num_inputs * np.prod(input_shape)
         weight_size = output_dim * np.prod(input_shape)

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

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

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

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

3 Affine layer: backward

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

```
In [14]: # 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 [16]: # 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 [17]: 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 has this problem, since there the range of activation is restricted to the range between 0 and 1 for all input. In the case of many layers stacked on top each other, this can produce remarkably little gradient flow.

6 "Sandwich" layers

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

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

```
In [18]: from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
         np.random.seed(231)
         x = np.random.randn(2, 3, 4)
         w = np.random.randn(12, 10)
         b = np.random.randn(10)
         dout = np.random.randn(2, 10)

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

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

         # Relative error should be around e-10 or less
         print('Testing affine_relu_forward and affine_relu_backward:')
         print('dx error: ', rel_error(dx_num, dx))
         print('dw error: ', rel_error(dw_num, dw))
         print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

7 Loss layers: Softmax and SVM

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

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

```
In [19]: 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 o
         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 arou
         print('\nTesting softmax_loss:')
         print('loss: ', loss)
         print('dx error: ', rel_error(dx_num, dx))
```

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

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

8 Two-layer network

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

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

```
In [22]: 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,
             [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,
             [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,
         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: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 3.12e-07
W2 relative error: 7.98e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

9 Solver

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

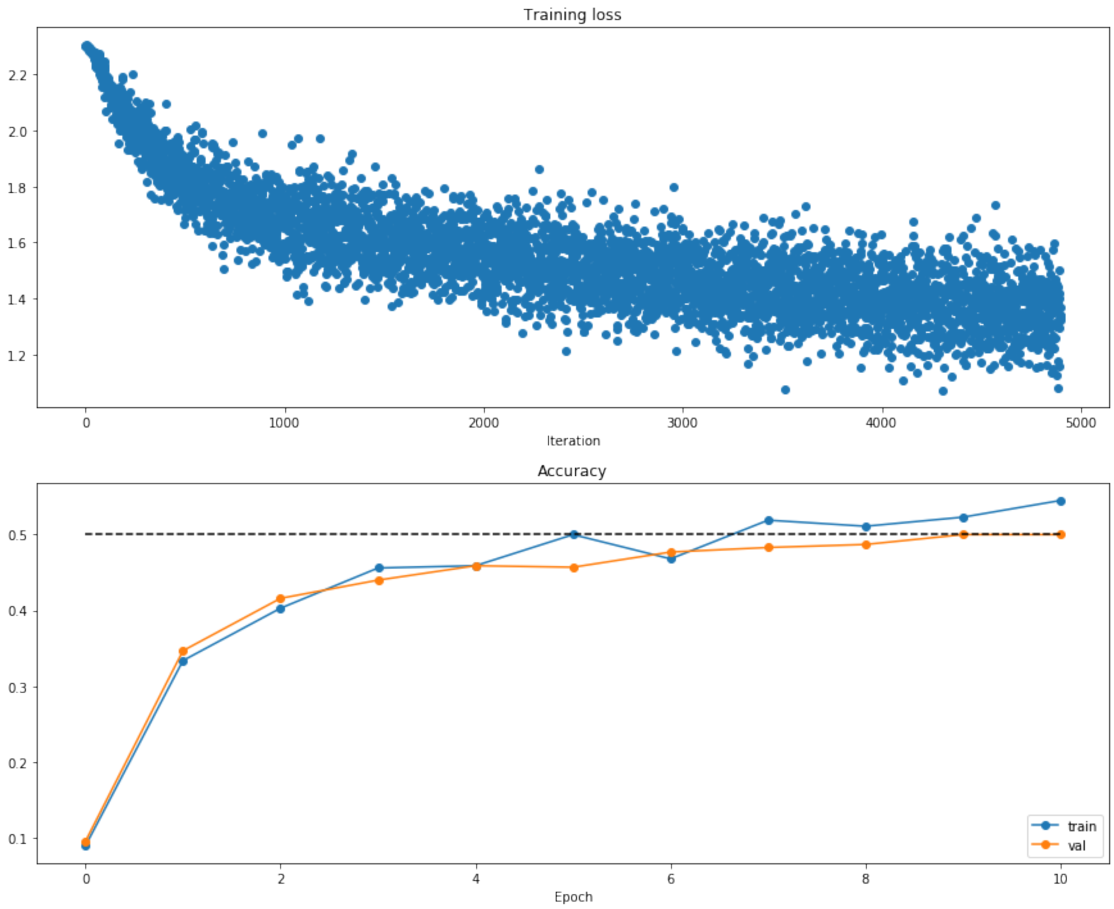
Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least 50% accuracy on the validation set.

```
In [24]: model = TwoLayerNet()
         solver = None

         ##############################################################################
         # TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
         # 50% accuracy on the validation set.                                        #
         ##############################################################################
         solver = Solver(model, data, print_every=100, optim_config={'learning_rate': 1.5e-4,}
         solver.train()
         ##############################################################################
         #                             END OF YOUR CODE                               #
         ##############################################################################
```

```
(Iteration 1 / 4900) loss: 2.304305
(Epoch 0 / 10) train acc: 0.090000; val_acc: 0.096000
(Iteration 101 / 4900) loss: 2.196478
(Iteration 201 / 4900) loss: 2.100096
(Iteration 301 / 4900) loss: 2.101613
(Iteration 401 / 4900) loss: 1.858383
(Epoch 1 / 10) train acc: 0.334000; val_acc: 0.347000
(Iteration 501 / 4900) loss: 1.876985
(Iteration 601 / 4900) loss: 1.769455
(Iteration 701 / 4900) loss: 1.713678
(Iteration 801 / 4900) loss: 1.570475
(Iteration 901 / 4900) loss: 1.734535
(Epoch 2 / 10) train acc: 0.403000; val_acc: 0.416000
(Iteration 1001 / 4900) loss: 1.669576
(Iteration 1101 / 4900) loss: 1.651928
(Iteration 1201 / 4900) loss: 1.445535
(Iteration 1301 / 4900) loss: 1.460460
(Iteration 1401 / 4900) loss: 1.499631
(Epoch 3 / 10) train acc: 0.456000; val_acc: 0.440000
(Iteration 1501 / 4900) loss: 1.571441
(Iteration 1601 / 4900) loss: 1.483204
(Iteration 1701 / 4900) loss: 1.683142
(Iteration 1801 / 4900) loss: 1.531149
(Iteration 1901 / 4900) loss: 1.542092
(Epoch 4 / 10) train acc: 0.459000; val_acc: 0.459000
(Iteration 2001 / 4900) loss: 1.695524
(Iteration 2101 / 4900) loss: 1.611489
(Iteration 2201 / 4900) loss: 1.640650
(Iteration 2301 / 4900) loss: 1.497137
(Iteration 2401 / 4900) loss: 1.531942
(Epoch 5 / 10) train acc: 0.500000; val_acc: 0.457000
(Iteration 2501 / 4900) loss: 1.280216
(Iteration 2601 / 4900) loss: 1.754380
(Iteration 2701 / 4900) loss: 1.401343
(Iteration 2801 / 4900) loss: 1.516286
(Iteration 2901 / 4900) loss: 1.586333
(Epoch 6 / 10) train acc: 0.468000; val_acc: 0.477000
(Iteration 3001 / 4900) loss: 1.641391
(Iteration 3101 / 4900) loss: 1.555542
(Iteration 3201 / 4900) loss: 1.438550
(Iteration 3301 / 4900) loss: 1.435463
(Iteration 3401 / 4900) loss: 1.460318
(Epoch 7 / 10) train acc: 0.519000; val_acc: 0.483000
```

```
(Iteration 3501 / 4900) loss: 1.481822
(Iteration 3601 / 4900) loss: 1.264959
(Iteration 3701 / 4900) loss: 1.274018
(Iteration 3801 / 4900) loss: 1.314330
(Iteration 3901 / 4900) loss: 1.442794
(Epoch 8 / 10) train acc: 0.511000; val_acc: 0.487000
(Iteration 4001 / 4900) loss: 1.555878
(Iteration 4101 / 4900) loss: 1.424935
(Iteration 4201 / 4900) loss: 1.349083
(Iteration 4301 / 4900) loss: 1.362923
(Iteration 4401 / 4900) loss: 1.392380
(Epoch 9 / 10) train acc: 0.523000; val_acc: 0.500000
(Iteration 4501 / 4900) loss: 1.418916
(Iteration 4601 / 4900) loss: 1.467329
(Iteration 4701 / 4900) loss: 1.413315
(Iteration 4801 / 4900) loss: 1.277754
(Epoch 10 / 10) train acc: 0.545000; val_acc: 0.500000
```

```
In [25]: # Run this cell to visualize training loss and train / val accuracy

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

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



10 Multilayer network

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

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

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

10.1 Initial loss and gradient check

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

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

```
In [46]: 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,))
```

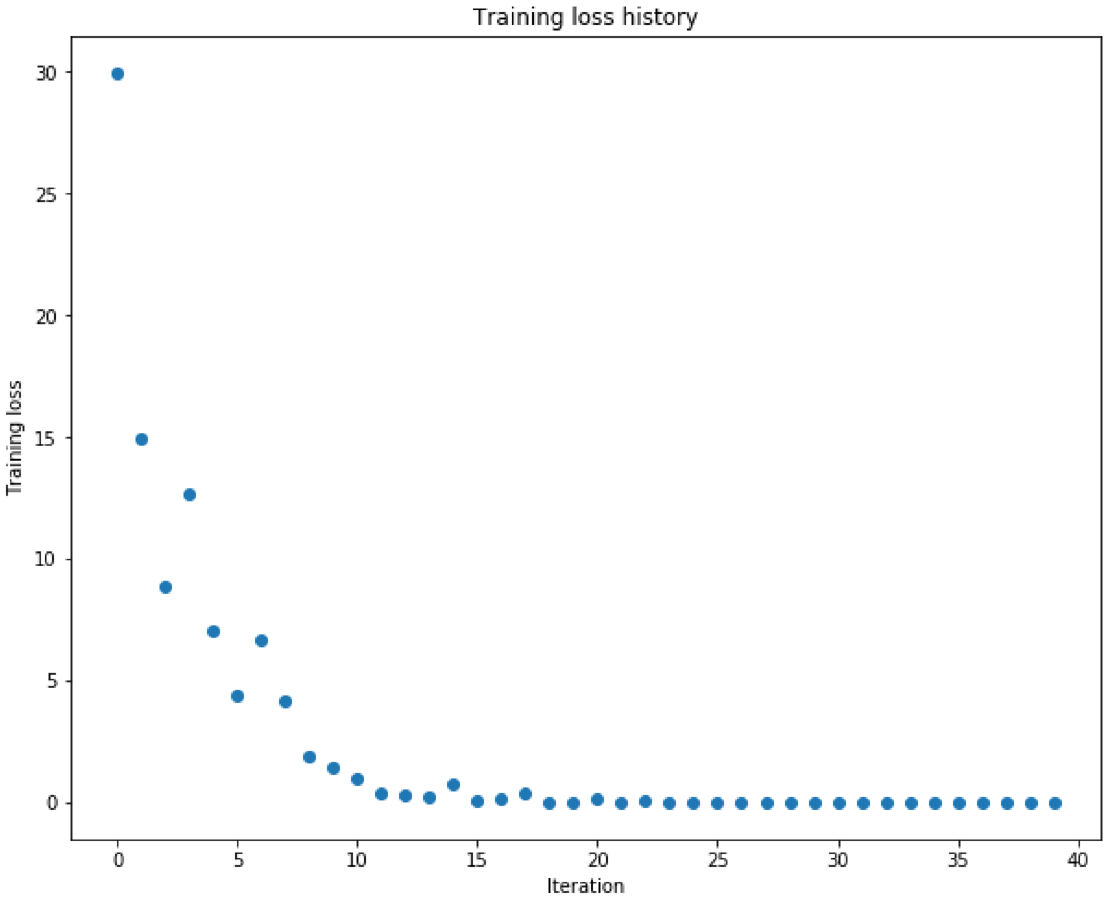
```
for reg in [0, 3.14]:
    print('Running check with reg = ', reg)
    model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                              reg=reg, weight_scale=5e-2, dtype=np.float64)

    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:  3.4046031324121993
Running check with reg =  3.14
Initial loss:  10.507945513778827
W1 relative error: 5.23e-08
W2 relative error: 6.36e-08
W3 relative error: 6.62e-08
b1 relative error: 5.05e-06
b2 relative error: 8.56e-09
b3 relative error: 1.07e-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 [60]: # 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'],
         }

         learning_rate = 1e-3
         weight_scale = 4e-2
         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()
```

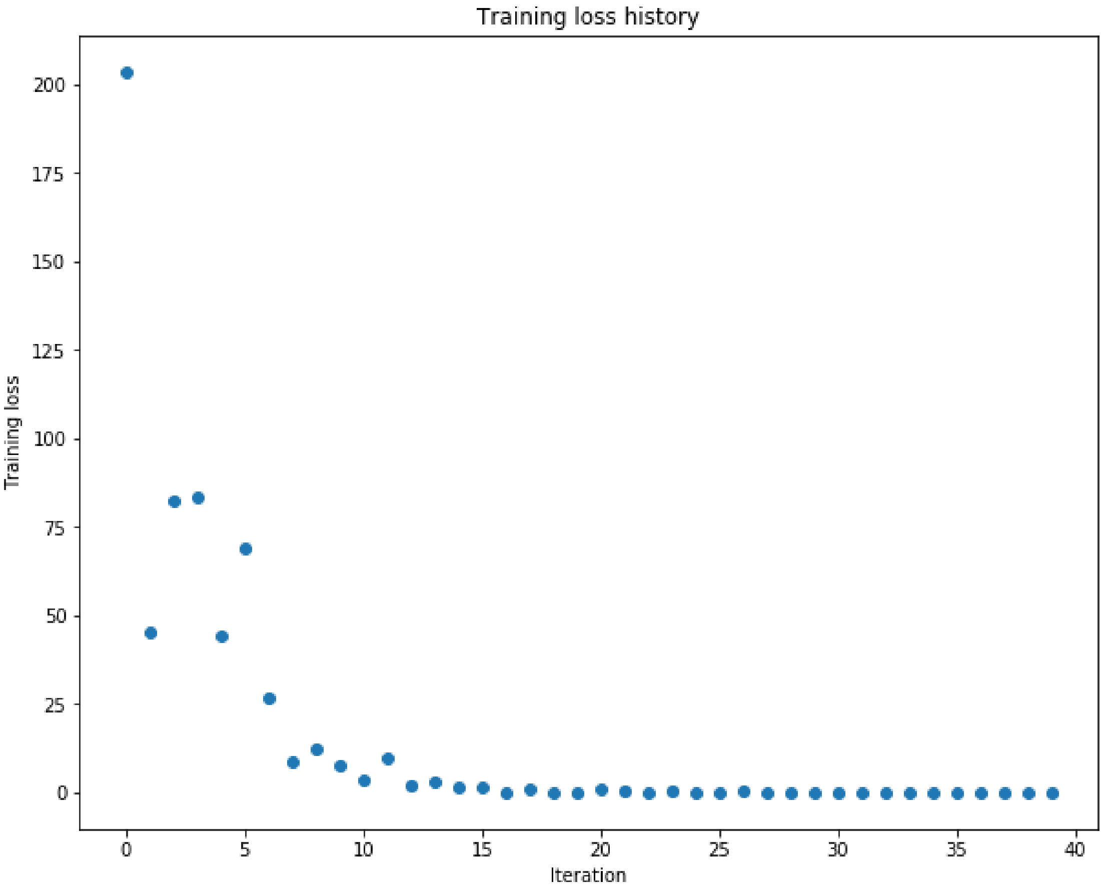
```
(Iteration 1 / 40) loss: 29.977639
(Epoch 0 / 20) train acc: 0.160000; val_acc: 0.047000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.056000
(Epoch 2 / 20) train acc: 0.400000; val_acc: 0.068000
(Epoch 3 / 20) train acc: 0.340000; val_acc: 0.107000
(Epoch 4 / 20) train acc: 0.680000; val_acc: 0.105000
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.108000
(Iteration 11 / 40) loss: 0.971397
(Epoch 6 / 20) train acc: 0.880000; val_acc: 0.117000
(Epoch 7 / 20) train acc: 0.940000; val_acc: 0.116000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.109000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.102000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.103000
(Iteration 21 / 40) loss: 0.130552
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.102000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.110000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.103000
(Iteration 31 / 40) loss: 0.005030
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.104000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.106000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.108000
```



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 [64]: # TODO: Use a five-layer Net to overfit 50 training examples by
         # tweaking just the learning rate and initialization scale.

         num_train = 50
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': data['y_val'],
         }

         learning_rate = 5e-4
         weight_scale = 1e-1
         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: 203.602260
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.078000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.099000
(Epoch 2 / 20) train acc: 0.340000; val_acc: 0.106000
(Epoch 3 / 20) train acc: 0.360000; val_acc: 0.108000
(Epoch 4 / 20) train acc: 0.520000; val_acc: 0.091000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.090000
(Iteration 11 / 40) loss: 3.499418
(Epoch 6 / 20) train acc: 0.800000; val_acc: 0.094000
(Epoch 7 / 20) train acc: 0.900000; val_acc: 0.103000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.107000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.108000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.108000
(Iteration 21 / 40) loss: 1.086958
(Epoch 11 / 20) train acc: 0.960000; val_acc: 0.102000
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.101000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.101000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.102000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.103000
(Iteration 31 / 40) loss: 0.001541
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.103000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.103000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.103000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.102000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.102000
```



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 5-layer net was more sensitive to initialization scale than the 3-layer net. This made tuning the 5-layer net more difficult. I think this is the case because more layers allows more susceptibility to exploding/disappearing gradients, due to a magnification of initial conditions occuring at every 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 http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file cs231n/optim.py and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function sgd_momentum and run the following to check your implementation. You should see errors less than e-8.

```
In [65]: from cs231n.optim import sgd_momentum

         N, D = 4, 5
         w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
         dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
         v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

         config = {'learning_rate': 1e-3, 'velocity': v}
         next_w, _ = sgd_momentum(w, dw, config=config)

         expected_next_w = np.asarray([
           [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
           [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
           [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
           [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
         expected_velocity = np.asarray([
           [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
           [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
           [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
           [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

         # Should see relative errors around e-8 or less
         print('next_w error: ', rel_error(next_w, expected_next_w))
         print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

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

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

```
In [66]: num_train = 4000
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': data['y_val'],
         }
```

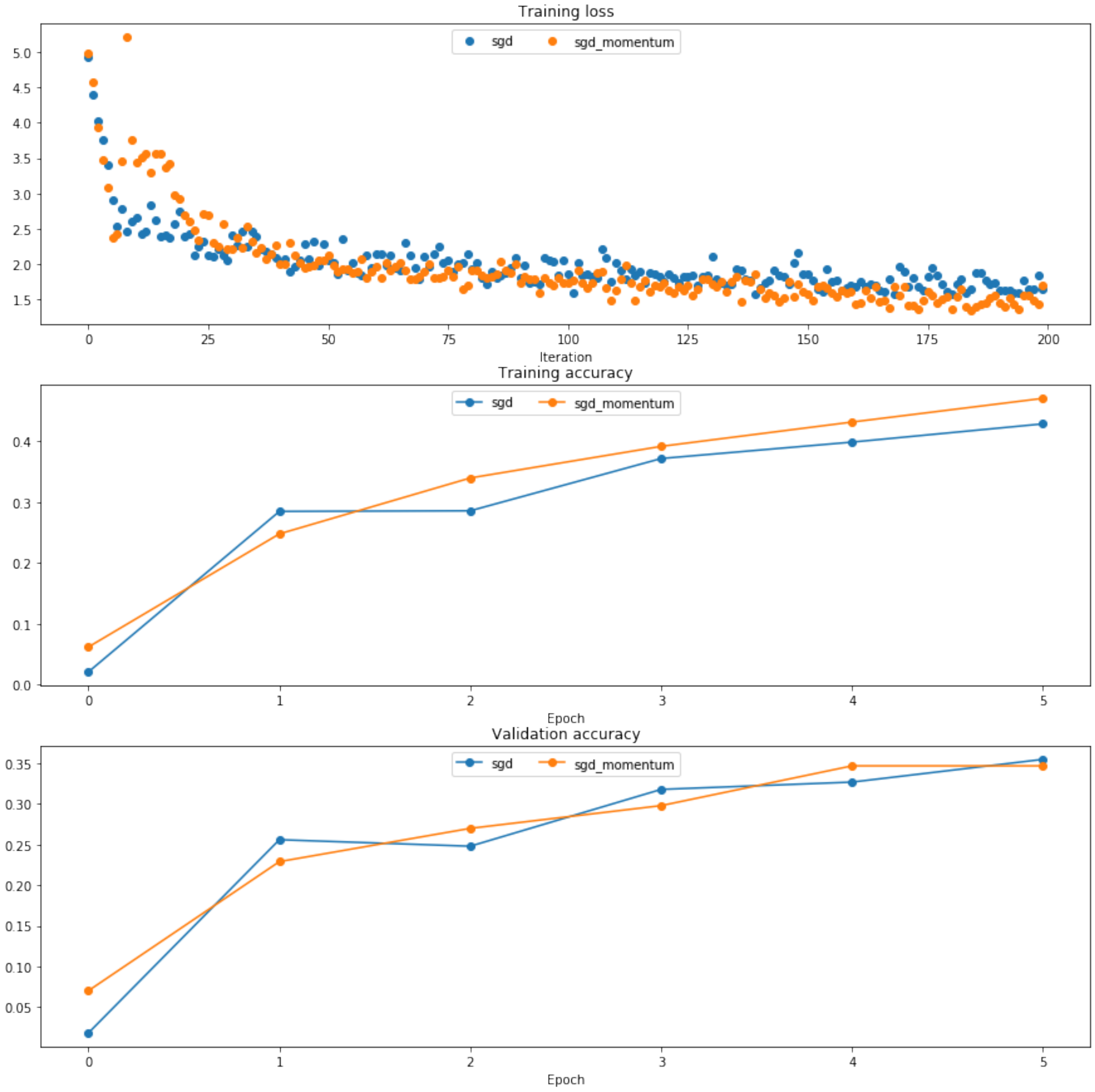
```
solvers = {}

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

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

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

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

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

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

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

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

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

```
running with  sgd
(Iteration 1 / 200) loss: 4.926570
```

```
(Epoch 0 / 5) train acc: 0.021000; val_acc: 0.018000
(Iteration 11 / 200) loss: 2.662493
(Iteration 21 / 200) loss: 2.402916
(Iteration 31 / 200) loss: 2.413534
(Epoch 1 / 5) train acc: 0.285000; val_acc: 0.256000
(Iteration 41 / 200) loss: 2.021112
(Iteration 51 / 200) loss: 2.054510
(Iteration 61 / 200) loss: 2.141004
(Iteration 71 / 200) loss: 2.109050
(Epoch 2 / 5) train acc: 0.286000; val_acc: 0.248000
(Iteration 81 / 200) loss: 1.929846
(Iteration 91 / 200) loss: 1.804004
(Iteration 101 / 200) loss: 1.864648
(Iteration 111 / 200) loss: 2.021089
(Epoch 3 / 5) train acc: 0.372000; val_acc: 0.318000
(Iteration 121 / 200) loss: 1.762394
(Iteration 131 / 200) loss: 2.106281
(Iteration 141 / 200) loss: 1.668880
(Iteration 151 / 200) loss: 1.860776
(Epoch 4 / 5) train acc: 0.399000; val_acc: 0.327000
(Iteration 161 / 200) loss: 1.682086
(Iteration 171 / 200) loss: 1.904640
(Iteration 181 / 200) loss: 1.582365
(Iteration 191 / 200) loss: 1.632679
(Epoch 5 / 5) train acc: 0.429000; val_acc: 0.355000

running with  sgd_momentum
(Iteration 1 / 200) loss: 4.983679
(Epoch 0 / 5) train acc: 0.062000; val_acc: 0.070000
(Iteration 11 / 200) loss: 3.440652
(Iteration 21 / 200) loss: 2.695438
(Iteration 31 / 200) loss: 2.215888
(Epoch 1 / 5) train acc: 0.248000; val_acc: 0.229000
(Iteration 41 / 200) loss: 2.008598
(Iteration 51 / 200) loss: 2.125732
(Iteration 61 / 200) loss: 1.961575
(Iteration 71 / 200) loss: 1.900730
(Epoch 2 / 5) train acc: 0.340000; val_acc: 0.270000
(Iteration 81 / 200) loss: 1.922797
(Iteration 91 / 200) loss: 1.743636
(Iteration 101 / 200) loss: 1.735316
(Iteration 111 / 200) loss: 1.635237
(Epoch 3 / 5) train acc: 0.392000; val_acc: 0.298000
(Iteration 121 / 200) loss: 1.745485
(Iteration 131 / 200) loss: 1.726241
(Iteration 141 / 200) loss: 1.649733
(Iteration 151 / 200) loss: 1.572955
(Epoch 4 / 5) train acc: 0.432000; val_acc: 0.347000
```

```
(Iteration 161 / 200) loss: 1.439884
(Iteration 171 / 200) loss: 1.683430
(Iteration 181 / 200) loss: 1.367654
(Iteration 191 / 200) loss: 1.456697
(Epoch 5 / 5) train acc: 0.471000; val_acc: 0.347000
```

```
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/matplotlib/cbook/depreca
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



13 RMSProp and Adam

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

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

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

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

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

```
In [73]: # Test RMSProp implementation
         from cs231n.optim import rmsprop

         N, D = 4, 5
         w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
         dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
         cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

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

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

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

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

```
In [70]: # Test Adam implementation
         from cs231n.optim import adam

         N, D = 4, 5
         w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
         dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
         m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
         v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

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

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

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

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

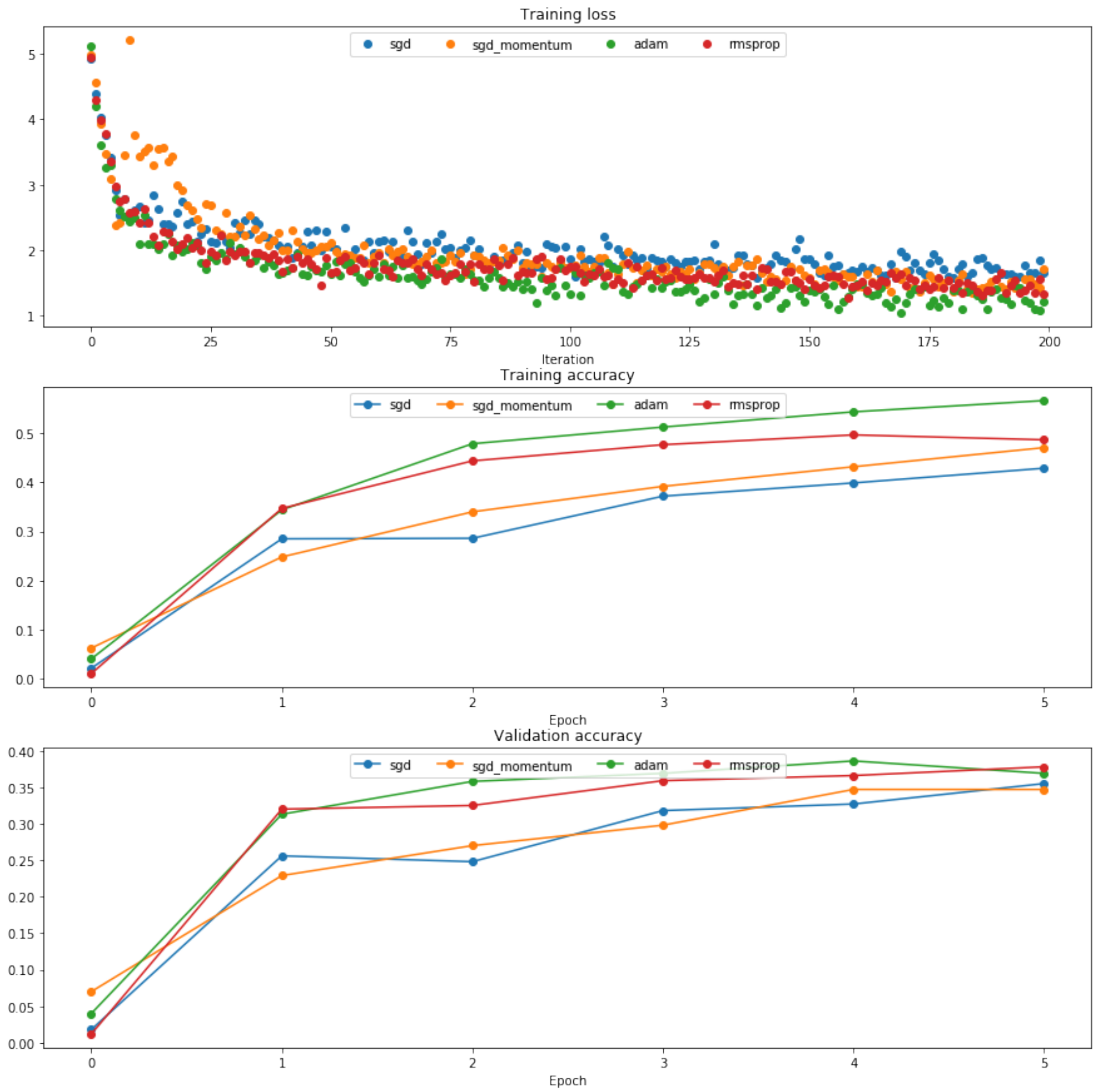
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```
In [74]: 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: 5.115555
(Epoch 0 / 5) train acc: 0.040000; val_acc: 0.040000
(Iteration 11 / 200) loss: 2.091939
(Iteration 21 / 200) loss: 2.029836
(Iteration 31 / 200) loss: 1.936269
(Epoch 1 / 5) train acc: 0.344000; val_acc: 0.313000
(Iteration 41 / 200) loss: 1.652198
(Iteration 51 / 200) loss: 1.647636
(Iteration 61 / 200) loss: 1.605885
(Iteration 71 / 200) loss: 1.558494
(Epoch 2 / 5) train acc: 0.479000; val_acc: 0.358000
(Iteration 81 / 200) loss: 1.538952
(Iteration 91 / 200) loss: 1.415405
(Iteration 101 / 200) loss: 1.341349
```

```
(Iteration 111 / 200) loss: 1.705892
(Epoch 3 / 5) train acc: 0.513000; val_acc: 0.369000
(Iteration 121 / 200) loss: 1.520427
(Iteration 131 / 200) loss: 1.339827
(Iteration 141 / 200) loss: 1.436781
(Iteration 151 / 200) loss: 1.384772
(Epoch 4 / 5) train acc: 0.544000; val_acc: 0.386000
(Iteration 161 / 200) loss: 1.492349
(Iteration 171 / 200) loss: 1.207942
(Iteration 181 / 200) loss: 1.389581
(Iteration 191 / 200) loss: 1.276552
(Epoch 5 / 5) train acc: 0.567000; val_acc: 0.369000

running with  rmsprop
(Iteration 1 / 200) loss: 4.954207
(Epoch 0 / 5) train acc: 0.010000; val_acc: 0.012000
(Iteration 11 / 200) loss: 2.429214
(Iteration 21 / 200) loss: 2.202003
(Iteration 31 / 200) loss: 1.926507
(Epoch 1 / 5) train acc: 0.347000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.674971
(Iteration 51 / 200) loss: 1.667160
(Iteration 61 / 200) loss: 1.923319
(Iteration 71 / 200) loss: 1.784594
(Epoch 2 / 5) train acc: 0.444000; val_acc: 0.325000
(Iteration 81 / 200) loss: 1.519633
(Iteration 91 / 200) loss: 1.677392
(Iteration 101 / 200) loss: 1.760079
(Iteration 111 / 200) loss: 1.561710
(Epoch 3 / 5) train acc: 0.477000; val_acc: 0.359000
(Iteration 121 / 200) loss: 1.567530
(Iteration 131 / 200) loss: 1.519366
(Iteration 141 / 200) loss: 1.733453
(Iteration 151 / 200) loss: 1.553716
(Epoch 4 / 5) train acc: 0.497000; val_acc: 0.366000
(Iteration 161 / 200) loss: 1.471567
(Iteration 171 / 200) loss: 1.470932
(Iteration 181 / 200) loss: 1.427699
(Iteration 191 / 200) loss: 1.650759
(Epoch 5 / 5) train acc: 0.487000; val_acc: 0.378000
```

```
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/matplotlib/cbook/depreca
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



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:

AdaGrad is likely overestimating the update direction of every parameter, therefore overshooting its update each time. Adam would not have this problem because it counters AdaGrad with

RMSProp.

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 [150]: best_model = None
          ################################################################################
          # TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
          # find batch/layer normalization and dropout useful. Store your best model in  #
          # the best_model variable.                                                     #
          ################################################################################
          X_val = data['X_val']
          y_val = data['y_val']
          X_test = data['X_test']
          y_test = data['y_test']

          lr = 1e-1
          ws = 2e-02

          model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=ws, dtype=np.float64, normalization=None, reg=1e-1)
          solver = Solver(model, data,
                  print_every=100, num_epochs=10, batch_size=100,
                  update_rule='sgd',
                  optim_config={
                    'learning_rate': lr,
                  },
                  lr_decay = 0.9,
                  verbose = True
                  )

          solver.train()

          best_model = model
          ################################################################################
          #                               END OF YOUR CODE                               #
          ################################################################################
```

```
(Iteration 1 / 4900) loss: 11.527302
```

```
(Epoch 0 / 10) train acc: 0.126000; val_acc: 0.131000
(Iteration 101 / 4900) loss: 6.912709
(Iteration 201 / 4900) loss: 3.168264
(Iteration 301 / 4900) loss: 2.423800
(Iteration 401 / 4900) loss: 2.381596
(Epoch 1 / 10) train acc: 0.174000; val_acc: 0.156000
(Iteration 501 / 4900) loss: 2.342675
(Iteration 601 / 4900) loss: 2.442615
(Iteration 701 / 4900) loss: 2.440142
(Iteration 801 / 4900) loss: 2.346380
(Iteration 901 / 4900) loss: 2.243061
(Epoch 2 / 10) train acc: 0.148000; val_acc: 0.138000
(Iteration 1001 / 4900) loss: 2.349707
(Iteration 1101 / 4900) loss: 2.477626
(Iteration 1201 / 4900) loss: 2.250109
(Iteration 1301 / 4900) loss: 2.624974
(Iteration 1401 / 4900) loss: 2.267361
(Epoch 3 / 10) train acc: 0.167000; val_acc: 0.144000
(Iteration 1501 / 4900) loss: 2.496892
(Iteration 1601 / 4900) loss: 2.210993
(Iteration 1701 / 4900) loss: 2.510397
(Iteration 1801 / 4900) loss: 2.269337
(Iteration 1901 / 4900) loss: 2.209090
(Epoch 4 / 10) train acc: 0.174000; val_acc: 0.156000
(Iteration 2001 / 4900) loss: 2.192810
(Iteration 2101 / 4900) loss: 2.105037
(Iteration 2201 / 4900) loss: 2.321909
(Iteration 2301 / 4900) loss: 2.223344
(Iteration 2401 / 4900) loss: 2.087857
(Epoch 5 / 10) train acc: 0.202000; val_acc: 0.196000
(Iteration 2501 / 4900) loss: 2.261432
(Iteration 2601 / 4900) loss: 2.291358
(Iteration 2701 / 4900) loss: 2.310307
(Iteration 2801 / 4900) loss: 2.187100
(Iteration 2901 / 4900) loss: 2.237614
(Epoch 6 / 10) train acc: 0.227000; val_acc: 0.182000
(Iteration 3001 / 4900) loss: 2.290527
(Iteration 3101 / 4900) loss: 2.123376
(Iteration 3201 / 4900) loss: 2.281698
(Iteration 3301 / 4900) loss: 2.316988
(Iteration 3401 / 4900) loss: 2.207121
(Epoch 7 / 10) train acc: 0.173000; val_acc: 0.174000
(Iteration 3501 / 4900) loss: 2.223589
(Iteration 3601 / 4900) loss: 2.312190
(Iteration 3701 / 4900) loss: 2.216970
(Iteration 3801 / 4900) loss: 2.203927
(Iteration 3901 / 4900) loss: 2.140064
(Epoch 8 / 10) train acc: 0.224000; val_acc: 0.213000
```

```
(Iteration 4001 / 4900) loss: 2.136392
(Iteration 4101 / 4900) loss: 2.147581
(Iteration 4201 / 4900) loss: 2.266644
(Iteration 4301 / 4900) loss: 2.217023
(Iteration 4401 / 4900) loss: 2.213490
(Epoch 9 / 10) train acc: 0.225000; val_acc: 0.267000
(Iteration 4501 / 4900) loss: 2.258316
(Iteration 4601 / 4900) loss: 2.140167
(Iteration 4701 / 4900) loss: 2.126551
(Iteration 4801 / 4900) loss: 2.063221
(Epoch 10 / 10) train acc: 0.246000; val_acc: 0.221000
```

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

```
Validation set accuracy:  0.267
Test set accuracy:  0.214
```

BatchNormalization

May 4, 2018

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

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

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

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

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

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

```
In [15]: # 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 cs231n/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 [16]: # 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:  [ 3.10862447e-17  7.27196081e-17 -2.15799600e-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 [17]: # 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 [18]: # 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:  1.7029261167605239e-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 $X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}$, 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 [24]: 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)))
```

```
dx difference:  5.964155941709756e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.57x
```

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 `cs231n/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 `cs231n/layer_utils.py`. If you decide to do so, do it in the file `cs231n/classifiers/fc_net.py`.

```
In [51]: 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
        print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
    if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  3.426041492229793
W1 relative error: 3.13e-03
W2 relative error: 4.03e-06
W3 relative error: 1.24e-09
b1 relative error: 2.78e-09
b2 relative error: 5.55e-09
b3 relative error: 5.03e-10
beta1 relative error: 1.00e+00
beta2 relative error: 1.00e+00
gamma1 relative error: 1.00e+00
gamma2 relative error: 1.00e+00

Running check with reg =  3.14
Initial loss:  10.415347168016266
W1 relative error: 3.17e-04
W2 relative error: 1.41e-05
W3 relative error: 3.99e-07
b1 relative error: 1.78e-07
b2 relative error: 2.22e-08
b3 relative error: 1.16e-09
beta1 relative error: 1.00e+00
beta2 relative error: 1.00e+00
gamma1 relative error: 1.00e+00
gamma2 relative error: 1.00e+00
```

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 [50]: 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='ba
         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()
```

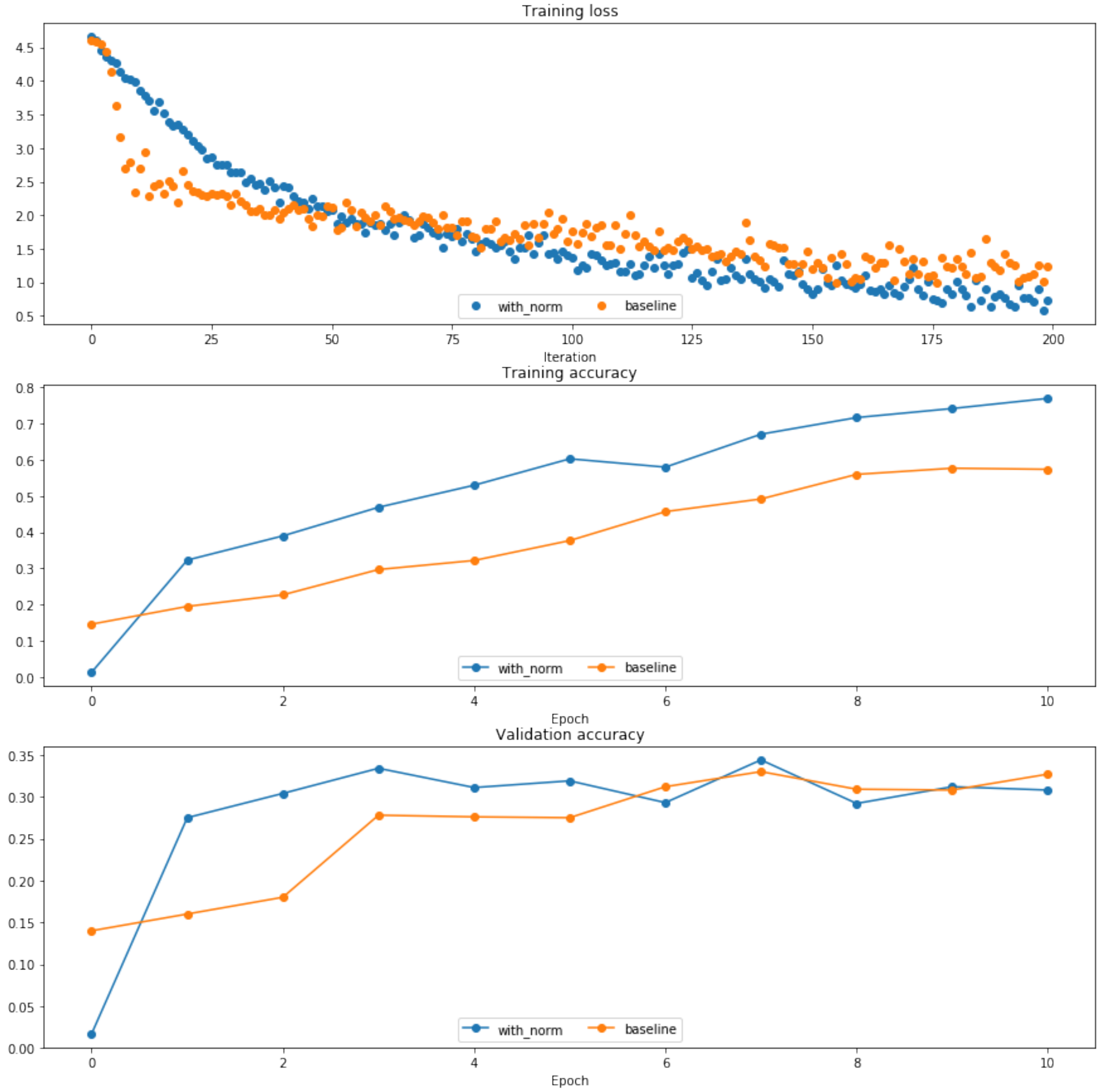
```
(Iteration 1 / 200) loss: 4.656364
(Epoch 0 / 10) train acc: 0.013000; val_acc: 0.017000
(Epoch 1 / 10) train acc: 0.323000; val_acc: 0.275000
(Iteration 21 / 200) loss: 3.209919
(Epoch 2 / 10) train acc: 0.390000; val_acc: 0.304000
(Iteration 41 / 200) loss: 2.443306
(Epoch 3 / 10) train acc: 0.469000; val_acc: 0.334000
(Iteration 61 / 200) loss: 1.882430
(Epoch 4 / 10) train acc: 0.530000; val_acc: 0.311000
(Iteration 81 / 200) loss: 1.466007
(Epoch 5 / 10) train acc: 0.603000; val_acc: 0.319000
(Iteration 101 / 200) loss: 1.359718
(Epoch 6 / 10) train acc: 0.580000; val_acc: 0.293000
(Iteration 121 / 200) loss: 1.122108
(Epoch 7 / 10) train acc: 0.671000; val_acc: 0.344000
(Iteration 141 / 200) loss: 0.915474
(Epoch 8 / 10) train acc: 0.717000; val_acc: 0.292000
(Iteration 161 / 200) loss: 0.983329
(Epoch 9 / 10) train acc: 0.742000; val_acc: 0.312000
(Iteration 181 / 200) loss: 1.016302
(Epoch 10 / 10) train acc: 0.770000; val_acc: 0.308000
(Iteration 1 / 200) loss: 4.603924
(Epoch 0 / 10) train acc: 0.146000; val_acc: 0.140000
(Epoch 1 / 10) train acc: 0.195000; val_acc: 0.160000
```

```
(Iteration 21 / 200) loss: 2.446764
(Epoch 2 / 10) train acc: 0.227000; val_acc: 0.180000
(Iteration 41 / 200) loss: 2.048511
(Epoch 3 / 10) train acc: 0.297000; val_acc: 0.278000
(Iteration 61 / 200) loss: 1.853403
(Epoch 4 / 10) train acc: 0.322000; val_acc: 0.276000
(Iteration 81 / 200) loss: 1.670860
(Epoch 5 / 10) train acc: 0.377000; val_acc: 0.275000
(Iteration 101 / 200) loss: 1.763908
(Epoch 6 / 10) train acc: 0.457000; val_acc: 0.312000
(Iteration 121 / 200) loss: 1.522823
(Epoch 7 / 10) train acc: 0.492000; val_acc: 0.330000
(Iteration 141 / 200) loss: 1.229490
(Epoch 8 / 10) train acc: 0.560000; val_acc: 0.309000
(Iteration 161 / 200) loss: 1.041544
(Epoch 9 / 10) train acc: 0.577000; val_acc: 0.308000
(Iteration 181 / 200) loss: 1.356394
(Epoch 10 / 10) train acc: 0.574000; val_acc: 0.327000
```

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 [52]: def plot_training_history(title, label, baseline, bn_solvers, plot_fn, bl_marker='.',
             """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

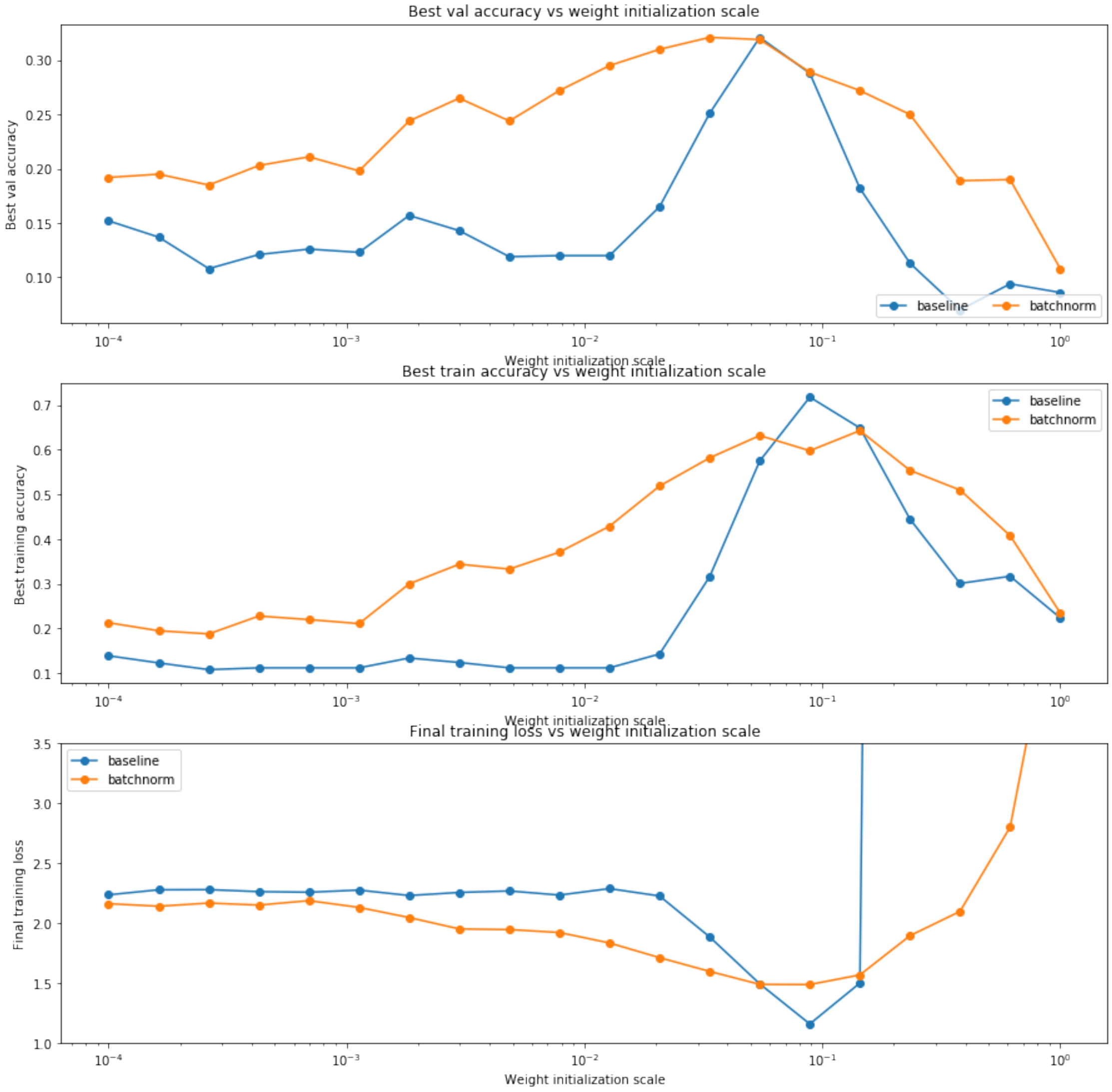
We will now run a small experiment to study the interaction of batch normalization and weight initialization.

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 [53]: 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=
           model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale, normalization=Non

           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 [54]: # 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:

Without batch normalzation, the final training loss and validation loss are highly sensitive to the weight initialization scale. Batch normalization causes uniformity of initialized weights, thereby reducing the effect of changes in feature space.

4 Batch normalization and batch size

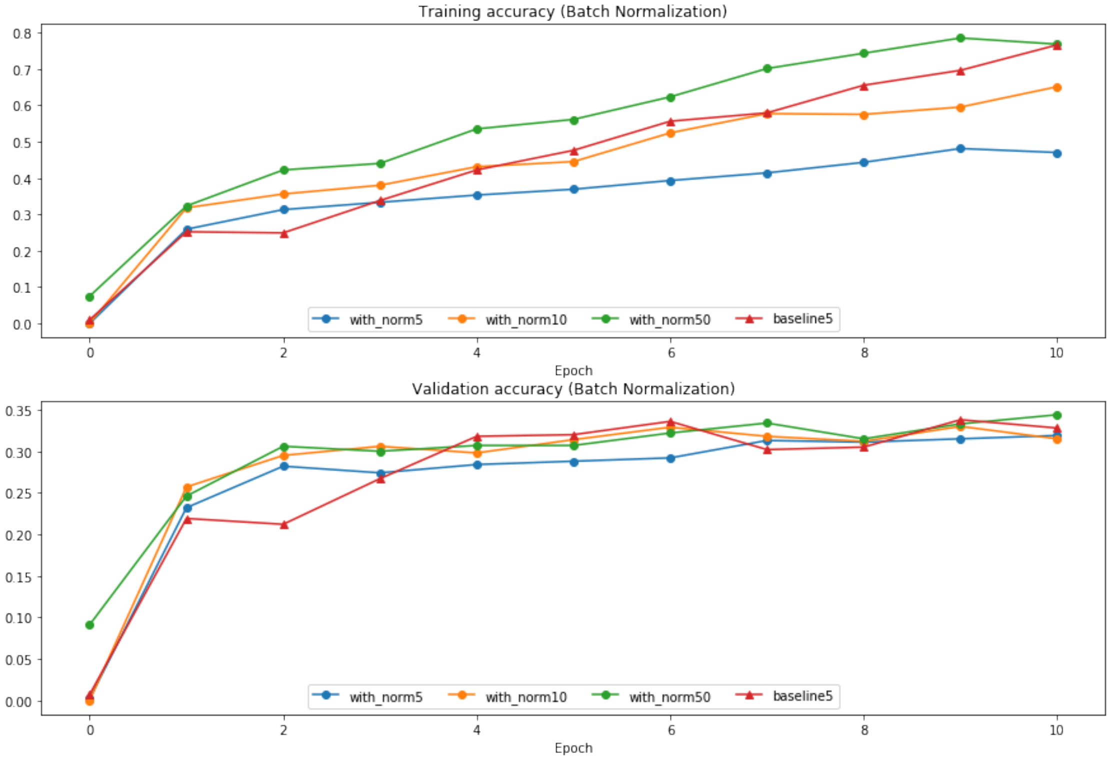
We will now run a small experiment to study the interaction of batch normalization and batch size.

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 [55]: 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=N
             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, normaliza
                 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 [56]: plt.subplot(2, 1, 1)
         plot_training_history('Training accuracy (Batch Normalization)','Epoch', solver_bsize
                               lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
         plt.subplot(2, 1, 2)
         plot_training_history('Validation accuracy (Batch Normalization)','Epoch', solver_bsi
                               lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lal

         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:

These results imply that batch normalization works well when batch size is large but not necessarily when batch size is small, as a small batch size actually performs worse than the baseline in these experiments.

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:

Option 3 = batch normalization
Option 2 = layer normalization

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 `cs231n/layers.py`, implement the forward pass for layer normalization in the function `layernorm_backward`.

Run the cell below to check your results. * In `cs231n/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cs231n/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 [64]: # 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:  [-0.58416774  0.03223092 -0.29106935  0.84300618]
  stds:   [0.42903404 1.0673565  1.17954475 0.38429471]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means:  [3.24749679 5.09669275 4.12679194 7.52901853]
  stds:   [1.28710213 3.2020695  3.53863425 1.15288413]
```

```
In [70]: # 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))
```

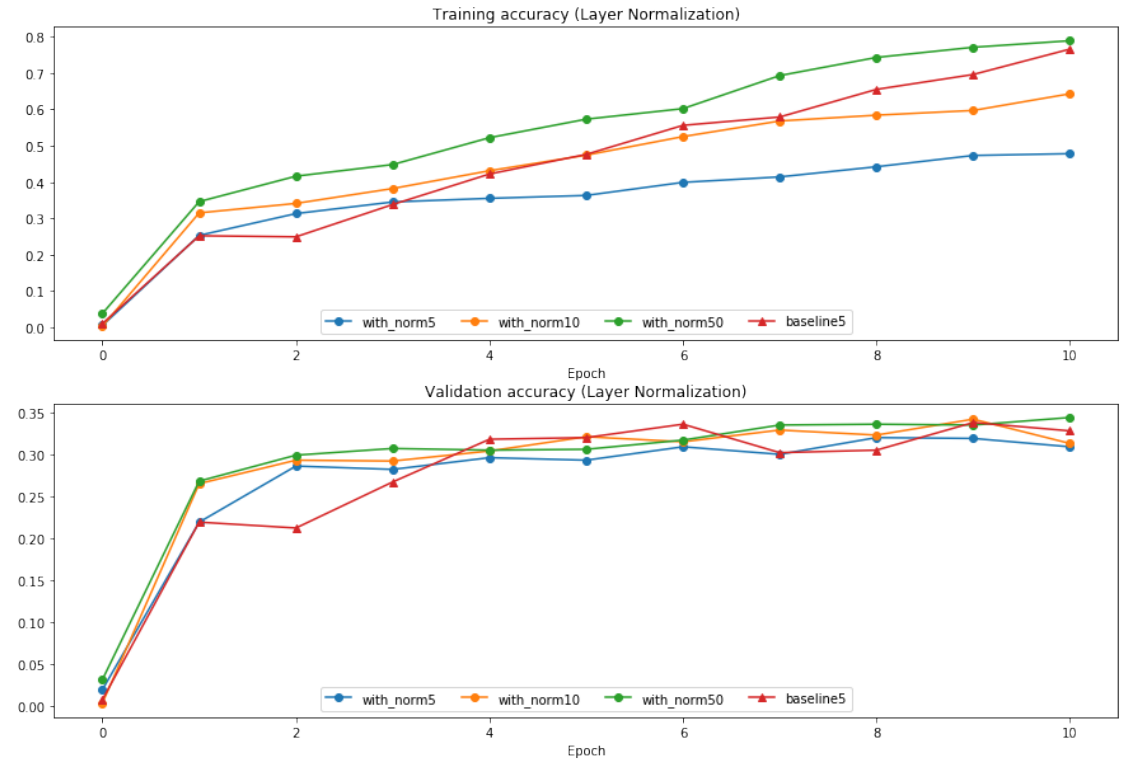
```
dx error:  1.7029261167605239e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

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 [72]: 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
                               lambda x: x.train_acc_history, bl_marker='-^', bn_marker='-o', l
         plt.subplot(2, 1, 2)
         plot_training_history('Validation accuracy (Layer Normalization)','Epoch', solver_bsiz
                               lambda x: x.val_acc_history, bl_marker='-^', bn_marker='-o', lab

         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:

2 + 3

Dropout

May 4, 2018

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 cs231n.classifiers.fc_net import *
        from cs231n.data_utils import get_CIFAR10_data
        from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arra
        from cs231n.solver import Solver

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

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

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

```
run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel
```

```
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 `cs231n/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 [8]: 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:  7.496693099233182
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.250216
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:  6.00904081497304
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.399204
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:  3.008739539965901
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.69926
Fraction of test-time output set to zero:  0.0
```

3 Dropout backward pass

In the file cs231n/layers.py, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

```
In [10]: 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)[

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

```
dx relative error:  1.892896957390533e-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:

4 Fully-connected nets with Dropout

In the file cs231n/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 [11]: 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:  3.4046031324121993
W1 relative error: 3.72e-07
W2 relative error: 8.64e-06
W3 relative error: 8.01e-07
b1 relative error: 2.10e-08
b2 relative error: 1.23e-09
b3 relative error: 3.94e-10

Running check with dropout =  0.75
Initial loss:  3.401186562367183
W1 relative error: 1.73e-08
W2 relative error: 3.07e-07
W3 relative error: 2.29e-05
b1 relative error: 5.22e-10
b2 relative error: 1.00e+00
b3 relative error: 2.58e-10

Running check with dropout =  0.5
Initial loss:  3.400803690996306
W1 relative error: 1.11e-06
W2 relative error: 8.84e-08
W3 relative error: 1.66e-06
b1 relative error: 2.08e-08
b2 relative error: 1.41e-09
b3 relative error: 3.57e-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.

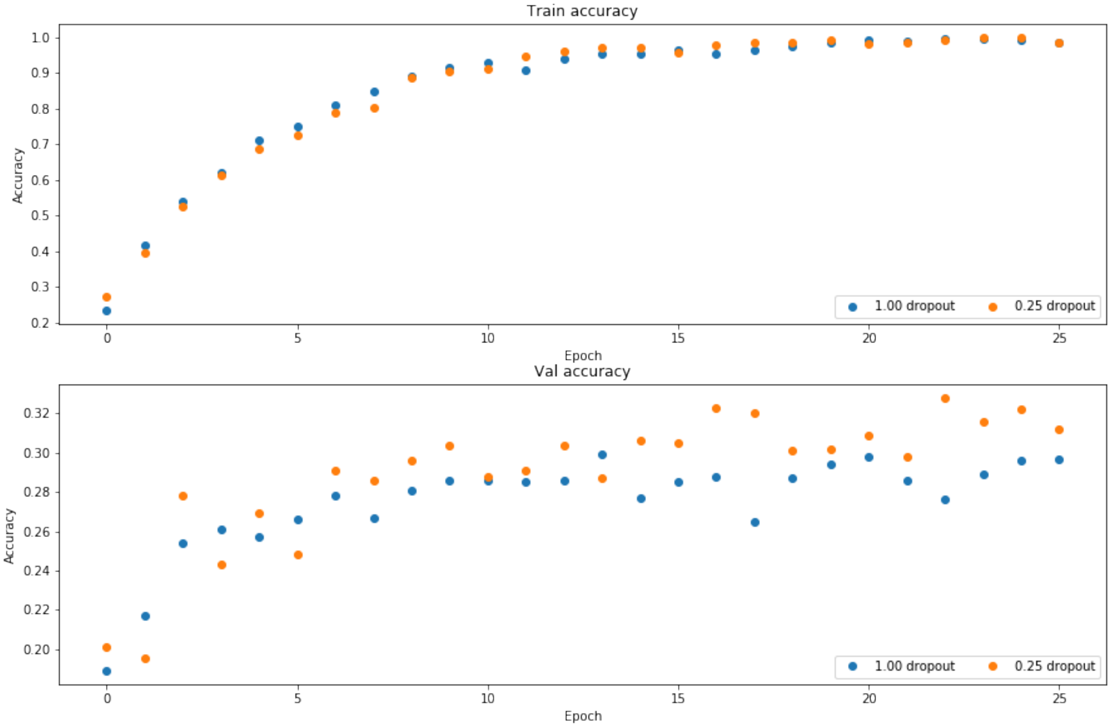
```
In [12]: # Train two identical nets, one with dropout and one without
         np.random.seed(231)
         num_train = 500
         small_data = {
           'X_train': data['X_train'][:num_train],
           'y_train': data['y_train'][:num_train],
           'X_val': data['X_val'],
           'y_val': 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: 18.076194
(Epoch 0 / 25) train acc: 0.234000; val_acc: 0.189000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.217000
(Epoch 2 / 25) train acc: 0.540000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.620000; val_acc: 0.261000
(Epoch 4 / 25) train acc: 0.712000; val_acc: 0.257000
(Epoch 5 / 25) train acc: 0.750000; val_acc: 0.266000
(Epoch 6 / 25) train acc: 0.810000; val_acc: 0.278000
(Epoch 7 / 25) train acc: 0.848000; val_acc: 0.267000
(Epoch 8 / 25) train acc: 0.888000; val_acc: 0.281000
(Epoch 9 / 25) train acc: 0.914000; val_acc: 0.286000
(Epoch 10 / 25) train acc: 0.928000; val_acc: 0.286000
(Epoch 11 / 25) train acc: 0.906000; val_acc: 0.285000
(Epoch 12 / 25) train acc: 0.938000; val_acc: 0.286000
(Epoch 13 / 25) train acc: 0.954000; val_acc: 0.299000
```

```
(Epoch 14 / 25) train acc: 0.952000; val_acc: 0.277000
(Epoch 15 / 25) train acc: 0.964000; val_acc: 0.285000
(Epoch 16 / 25) train acc: 0.952000; val_acc: 0.288000
(Epoch 17 / 25) train acc: 0.964000; val_acc: 0.265000
(Epoch 18 / 25) train acc: 0.974000; val_acc: 0.287000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.294000
(Epoch 20 / 25) train acc: 0.990000; val_acc: 0.298000
(Iteration 101 / 125) loss: 0.018257
(Epoch 21 / 25) train acc: 0.988000; val_acc: 0.286000
(Epoch 22 / 25) train acc: 0.994000; val_acc: 0.276000
(Epoch 23 / 25) train acc: 0.996000; val_acc: 0.289000
(Epoch 24 / 25) train acc: 0.990000; val_acc: 0.296000
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.297000
0.25
(Iteration 1 / 125) loss: 14.008363
(Epoch 0 / 25) train acc: 0.272000; val_acc: 0.201000
(Epoch 1 / 25) train acc: 0.394000; val_acc: 0.195000
(Epoch 2 / 25) train acc: 0.526000; val_acc: 0.278000
(Epoch 3 / 25) train acc: 0.612000; val_acc: 0.243000
(Epoch 4 / 25) train acc: 0.686000; val_acc: 0.269000
(Epoch 5 / 25) train acc: 0.726000; val_acc: 0.248000
(Epoch 6 / 25) train acc: 0.788000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.802000; val_acc: 0.286000
(Epoch 8 / 25) train acc: 0.886000; val_acc: 0.296000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.304000
(Epoch 10 / 25) train acc: 0.910000; val_acc: 0.288000
(Epoch 11 / 25) train acc: 0.946000; val_acc: 0.291000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.304000
(Epoch 13 / 25) train acc: 0.970000; val_acc: 0.287000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.306000
(Epoch 15 / 25) train acc: 0.958000; val_acc: 0.305000
(Epoch 16 / 25) train acc: 0.976000; val_acc: 0.323000
(Epoch 17 / 25) train acc: 0.986000; val_acc: 0.320000
(Epoch 18 / 25) train acc: 0.986000; val_acc: 0.301000
(Epoch 19 / 25) train acc: 0.990000; val_acc: 0.302000
(Epoch 20 / 25) train acc: 0.982000; val_acc: 0.309000
(Iteration 101 / 125) loss: 0.245039
(Epoch 21 / 25) train acc: 0.986000; val_acc: 0.298000
(Epoch 22 / 25) train acc: 0.992000; val_acc: 0.328000
(Epoch 23 / 25) train acc: 0.998000; val_acc: 0.316000
(Epoch 24 / 25) train acc: 0.998000; val_acc: 0.322000
(Epoch 25 / 25) train acc: 0.984000; val_acc: 0.312000
```

```
In [13]: # 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:

In our results, the training accuracies are the same but the validation accuracies are different. It seems that use of dropout does not increase training accuracy but it does increase validation accuracy.

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:

p should be decreased in proportion to the decrease of the size of the hidden layers.

ConvolutionalNetworks

May 4, 2018

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 cs231n.classifiers.cnn import *
        from cs231n.data_utils import get_CIFAR10_data
        from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradie
        from cs231n.layers import *
        from cs231n.fast_layers import *
        from cs231n.solver import Solver

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

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

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

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 `cs231n/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
```

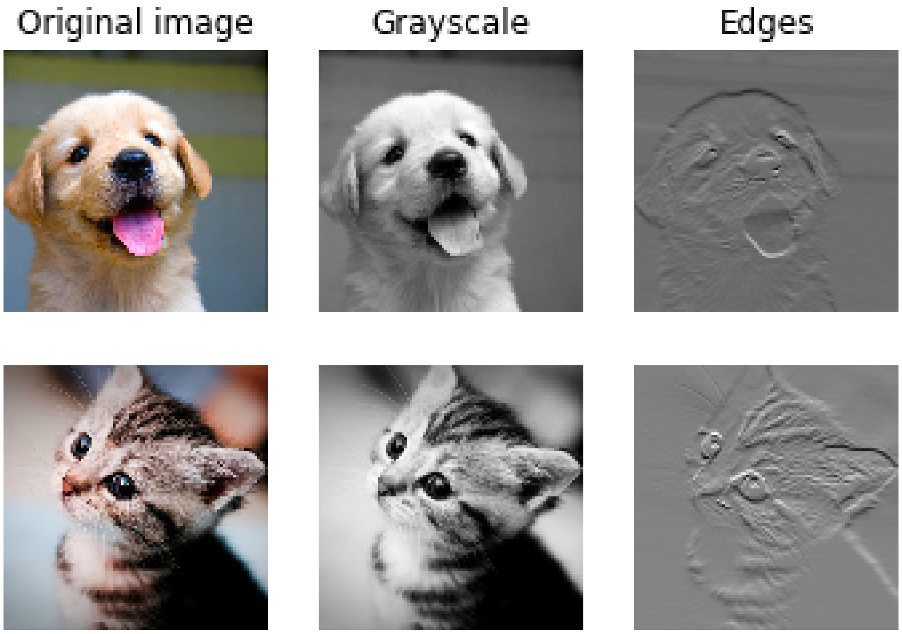
3 Aside: Image processing via convolutions

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 [4]: 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()
```

```
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/ipykernel_launcher.py:3
`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
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/ipykernel_launcher.py:1
`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.
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/ipykernel_launcher.py:1
`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 `cs231n/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 [5]: 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}

        dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param
        dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param
        db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param

        out, cache = conv_forward_naive(x, w, b, conv_param)
        dx, dw, db = conv_backward_naive(dout, cache)

        # 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:  1.7056541140727924e-08
dw error:  2.5239797851386593e-10
db error:  4.720827687262576e-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 `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
In [6]: 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 `cs231n/layers.py`. You don't need to worry about computational efficiency.

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

```
In [7]: 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)

        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 `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` 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 [8]: # Rel errors should be around e-9 or less
        from cs231n.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: 5.214998s
Fast: 0.038335s
Speedup: 136.038056x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 8.481010s
Fast: 0.033575s
Speedup: 252.600321x
dx difference:  9.43434568725122e-12
dw difference:  9.666193083607238e-13
db difference:  7.17573897727347e-15
```

```
In [9]: # Relative errors should be close to 0.0
        from cs231n.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: 0.379368s
fast: 0.002917s
speedup: 130.051900x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.475831s
fast: 0.013512s
speedup: 35.215711x
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 `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

```
In [10]: from cs231n.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_
         dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_
         db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_

         # 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:  6.514336569263308e-09
dw error:  1.437177519182391e-08
db error:  2.0587934544554346e-09
```

```
In [11]: from cs231n.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
         dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param
         db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param

         # 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:  3.5600610115232832e-09
```

```
dw error:  2.2848250390590822e-10
db error:  1.3087619975802167e-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 cs231n/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 [13]: 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)
```

```
Initial loss (no regularization):  2.30258318667065
Initial loss (with regularization):  2.508342473551143
```

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 [14]: 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=Fals
             e = rel_error(param_grad_num, grads[param_name])
             print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[p
```

```
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

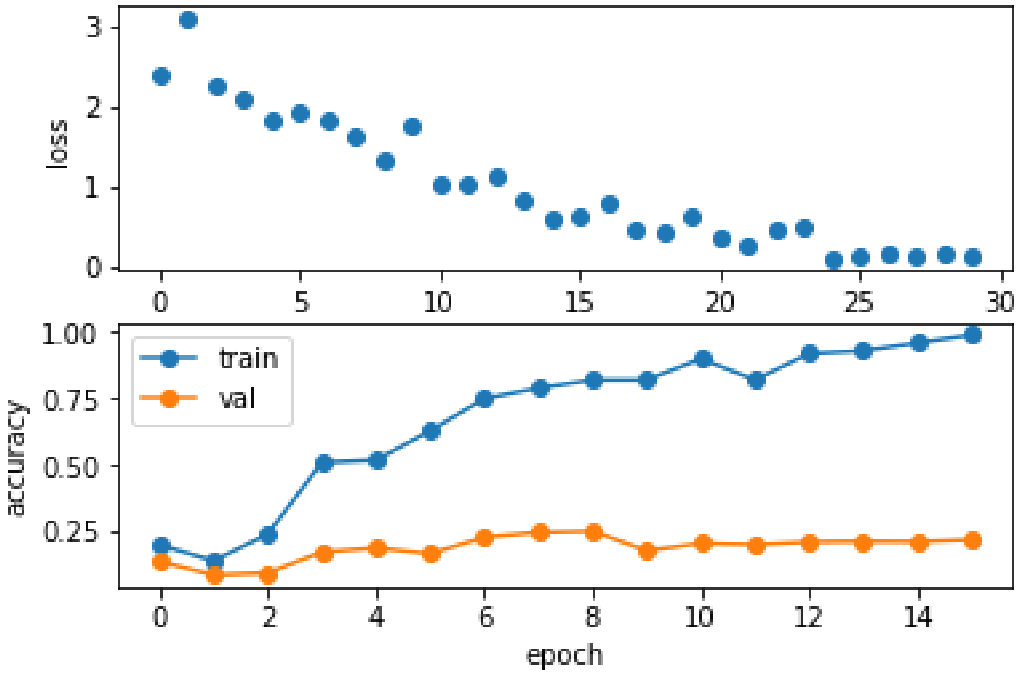
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
In [15]: 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=1e-2)

         solver = Solver(model, small_data,
                         num_epochs=15, batch_size=50,
                         update_rule='adam',
                         optim_config={
                           'learning_rate': 1e-3,
                         },
                         verbose=True, print_every=1)
         solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

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

```
In [16]: 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 [17]: model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

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

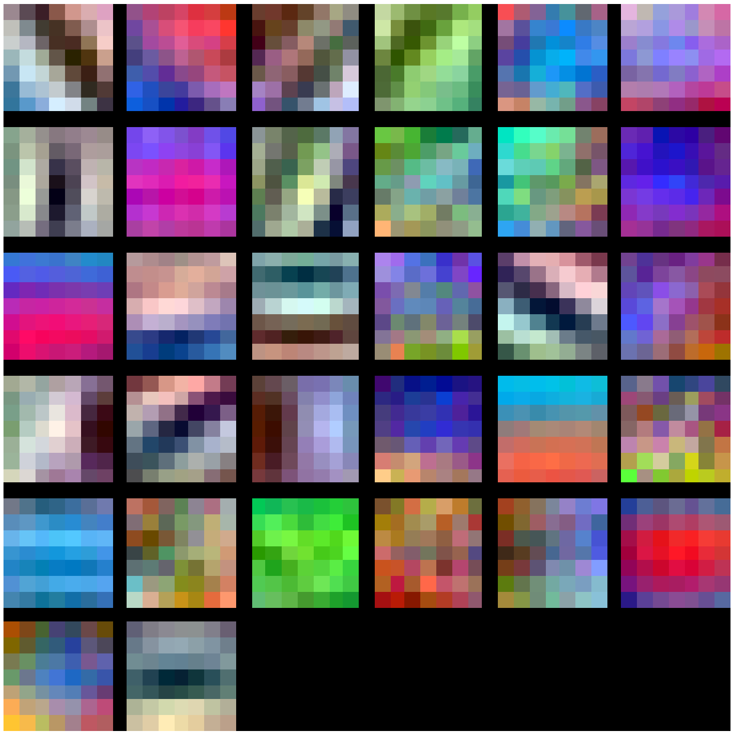
```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
```

```
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

9.5 Visualize Filters

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

```
In [18]: from cs231n.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 `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

```
In [19]: 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 [20]: 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 `cs231n/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 [21]: 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))
```

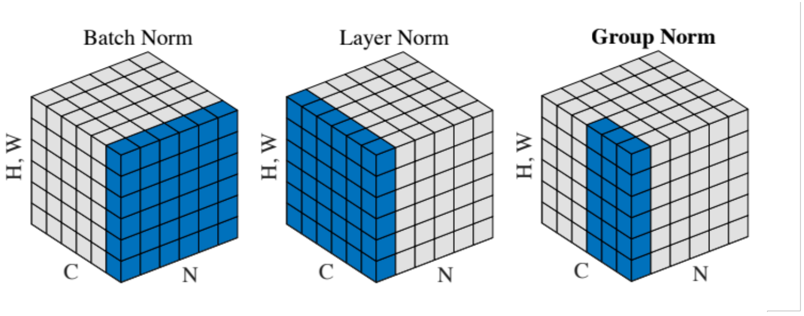
```
dx error:  2.7866481909597216e-07
dgamma error:  7.097288082068512e-12
dbeta error:  3.275494761332149e-12
```

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 cs231n/layers.py, implement the forward pass for group normalization in the function spatial_groupnorm_forward. Check your implementation by running the following:

```
In [23]: 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]
After spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [ 0.13335201 -0.0708835  -0.13335201  0.0708835 ]
  Stds:  [0.99083728 0.99376409 0.99083728 0.99376409]
```

11.2 Spatial group normalization: backward

In the file `cs231n/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 [24]: 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)

         _, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
         dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
         #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))
```

```
dx error:  0.0010103963079019869
dgamma error:  1.0
dbeta error:  1.0
```

TensorFlow

May 4, 2018

1 What's this TensorFlow business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, TensorFlow (or PyTorch, if you switch over to that notebook)

What is it? TensorFlow is a system for executing computational graphs over Tensor objects, with native support for performing backpropogation for its Variables. In it, we work with Tensors which are n-dimensional arrays analogous to the numpy ndarray.

Why?

- Our code will now run on GPUs! Much faster training. Writing your own modules to run on GPUs is beyond the scope of this class, unfortunately.
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.1 How will I learn TensorFlow?

TensorFlow has many excellent tutorials available, including those from Google themselves.

Otherwise, this notebook will walk you through much of what you need to do to train models in TensorFlow. See the end of the notebook for some links to helpful tutorials if you want to learn more or need further clarification on topics that aren't fully explained here.

2 Table of Contents

This notebook has 5 parts. We will walk through TensorFlow at three different levels of abstraction, which should help you better understand it and prepare you for working on your project.

1. Preparation: load the CIFAR-10 dataset.
2. Barebone TensorFlow: we will work directly with low-level TensorFlow graphs.
3. Keras Model API: we will use `tf.keras.Model` to define arbitrary neural network architecture.
4. Keras Sequential API: we will use `tf.keras.Sequential` to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`tf.keras.Model`	High	Medium
`tf.keras.Sequential`	Low	High

3 Part I: Preparation

First, we load the CIFAR-10 dataset. This might take a few minutes to download the first time you run it, but after that the files should be cached on disk and loading should be faster.

In previous parts of the assignment we used CS231N-specific code to download and read the CIFAR-10 dataset; however the `tf.keras.datasets` package in TensorFlow provides prebuilt utility functions for loading many common datasets.

For the purposes of this assignment we will still write our own code to preprocess the data and iterate through it in minibatches. The `tf.data` package in TensorFlow provides tools for automating this process, but working with this package adds extra complication and is beyond the scope of this notebook. However using `tf.data` can be much more efficient than the simple approach used in this notebook, so you should consider using it for your project.

```
In [2]: import os
        import tensorflow as tf
        import numpy as np
        import math
        import timeit
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```
/Users/ianscottknight/anaconda/envs/cs231n/lib/python3.6/site-packages/h5py/__init__.py:36: Fu
  from ._conv import register_converters as _register_converters
```

```
In [3]: def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
            """
            Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepare
            it for the two-layer neural net classifier. These are the same steps as
```

```
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean pixel and divide by std
    mean_pixel = X_train.mean(axis=(0, 1, 2), keepdims=True)
    std_pixel = X_train.std(axis=(0, 1, 2), keepdims=True)
    X_train = (X_train - mean_pixel) / std_pixel
    X_val = (X_val - mean_pixel) / std_pixel
    X_test = (X_test - mean_pixel) / std_pixel

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


# Invoke the above function to get our data.
NHW = (0, 1, 2)
X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape, y_train.dtype)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 87s 1us/step
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

3.0.1 Preparation: Dataset object

For our own convenience we'll define a lightweight `Dataset` class which lets us iterate over data and labels. This is not the most flexible or most efficient way to iterate through data, but it will serve our purposes.

```
In [4]: class Dataset(object):
            def __init__(self, X, y, batch_size, shuffle=False):
                """
                Construct a Dataset object to iterate over data X and labels y

                Inputs:
                - X: Numpy array of data, of any shape
                - y: Numpy array of labels, of any shape but with y.shape[0] == X.shape[0]
                - batch_size: Integer giving number of elements per minibatch
                - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
                """
                assert X.shape[0] == y.shape[0], 'Got different numbers of data and labels'
                self.X, self.y = X, y
                self.batch_size, self.shuffle = batch_size, shuffle

            def __iter__(self):
                N, B = self.X.shape[0], self.batch_size
                idxs = np.arange(N)
                if self.shuffle:
                    np.random.shuffle(idxs)
                return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))


        train_dset = Dataset(X_train, y_train, batch_size=64, shuffle=True)
        val_dset = Dataset(X_val, y_val, batch_size=64, shuffle=False)
        test_dset = Dataset(X_test, y_test, batch_size=64)
```

```
In [5]: # We can iterate through a dataset like this:
        for t, (x, y) in enumerate(train_dset):
            print(t, x.shape, y.shape)
            if t > 5: break
```

```
0 (64, 32, 32, 3) (64,)
1 (64, 32, 32, 3) (64,)
2 (64, 32, 32, 3) (64,)
3 (64, 32, 32, 3) (64,)
4 (64, 32, 32, 3) (64,)
5 (64, 32, 32, 3) (64,)
6 (64, 32, 32, 3) (64,)
```

You can optionally **use GPU by setting the flag to True below**. It's not neccessary to use a GPU for this assignment; if you are working on Google Cloud then we recommend that you do not use a GPU, as it will be significantly more expensive.

```
In [10]: # Set up some global variables
         USE_GPU = False

         if USE_GPU:
             device = '/device:GPU:0'
         else:
             device = '/cpu:0'

         # Constant to control how often we print when training models
         print_every = 100

         print('Using device: ', device)
```

```
Using device:  /cpu:0
```

4 Part II: Barebone TensorFlow

TensorFlow ships with various high-level APIs which make it very convenient to define and train neural networks; we will cover some of these constructs in Part III and Part IV of this notebook. In this section we will start by building a model with basic TensorFlow constructs to help you better understand what's going on under the hood of the higher-level APIs.

TensorFlow is primarily a framework for working with **static computational graphs**. Nodes in the computational graph are Tensors which will hold n-dimensional arrays when the graph is run; edges in the graph represent functions that will operate on Tensors when the graph is run to actually perform useful computation.

This means that a typical TensorFlow program is written in two distinct phases:

1. Build a computational graph that describes the computation that you want to perform. This stage doesn't actually perform any computation; it just builds up a symbolic representation of your computation. This stage will typically define one or more `placeholder` objects that represent inputs to the computational graph.
2. Run the computational graph many times. Each time the graph is run you will specify which parts of the graph you want to compute, and pass a `feed_dict` dictionary that will give concrete values to any `placeholders` in the graph.

4.0.1 TensorFlow warmup: Flatten Function

We can see this in action by defining a simple `flatten` function that will reshape image data for use in a fully-connected network.

In TensorFlow, data for convolutional feature maps is typically stored in a Tensor of shape N x H x W x C where:

- N is the number of datapoints (minibatch size)

- H is the height of the feature map
- W is the width of the feature map
- C is the number of channels in the feature map

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector -- it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `H x W x C` values per representation into a single long vector. The flatten function below first reads in the value of N from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be H x W x C, but we don't need to specify that explicitly).

NOTE: TensorFlow and PyTorch differ on the default Tensor layout; TensorFlow uses N x H x W x C but PyTorch uses N x C x H x W.

```
In [11]: def flatten(x):
             """
             Input:
             - TensorFlow Tensor of shape (N, D1, ..., DM)

             Output:
             - TensorFlow Tensor of shape (N, D1 * ... * DM)
             """
             N = tf.shape(x)[0]
             return tf.reshape(x, (N, -1))
```

```
In [12]: def test_flatten():
             # Clear the current TensorFlow graph.
             tf.reset_default_graph()

             # Stage I: Define the TensorFlow graph describing our computation.
             # In this case the computation is trivial: we just want to flatten
             # a Tensor using the flatten function defined above.

             # Our computation will have a single input, x. We don't know its
             # value yet, so we define a placeholder which will hold the value
             # when the graph is run. We then pass this placeholder Tensor to
             # the flatten function; this gives us a new Tensor which will hold
             # a flattened view of x when the graph is run. The tf.device
             # context manager tells TensorFlow whether to place these Tensors
             # on CPU or GPU.
             with tf.device(device):
                 x = tf.placeholder(tf.float32)
                 x_flat = flatten(x)

             # At this point we have just built the graph describing our computation,
             # but we haven't actually computed anything yet. If we print x and x_flat
```

```
    # we see that they don't hold any data; they are just TensorFlow Tensors
    # representing values that will be computed when the graph is run.
    print('x: ', type(x), x)
    print('x_flat: ', type(x_flat), x_flat)
    print()

    # We need to use a TensorFlow Session object to actually run the graph.
    with tf.Session() as sess:
        # Construct concrete values of the input data x using numpy
        x_np = np.arange(24).reshape((2, 3, 4))
        print('x_np:\n', x_np, '\n')

        # Run our computational graph to compute a concrete output value.
        # The first argument to sess.run tells TensorFlow which Tensor
        # we want it to compute the value of; the feed_dict specifies
        # values to plug into all placeholder nodes in the graph. The
        # resulting value of x_flat is returned from sess.run as a
        # numpy array.
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np, '\n')

        # We can reuse the same graph to perform the same computation
        # with different input data
        x_np = np.arange(12).reshape((2, 3, 2))
        print('x_np:\n', x_np, '\n')
        x_flat_np = sess.run(x_flat, feed_dict={x: x_np})
        print('x_flat_np:\n', x_flat_np)
test_flatten()
```

```
x:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Placeholder:0", dtype=float32, dev
x_flat:  <class 'tensorflow.python.framework.ops.Tensor'> Tensor("Reshape:0", shape=(?, ?), dty

x_np:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [12. 13. 14. 15. 16. 17. 18. 19. 20. 21. 22. 23.]]

x_np:
 [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]

x_flat_np:
 [[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]]
```

4.0.2 Barebones TensorFlow: Two-Layer Network

We will now implement our first neural network with TensorFlow: a fully-connected ReLU network with two hidden layers and no biases on the CIFAR10 dataset. For now we will use only low-level TensorFlow operators to define the network; later we will see how to use the higher-level abstractions provided by `tf.keras` to simplify the process.

We will define the forward pass of the network in the function `two_layer_fc`; this will accept TensorFlow Tensors for the inputs and weights of the network, and return a TensorFlow Tensor for the scores. It's important to keep in mind that calling the `two_layer_fc` function **does not** perform any computation; instead it just sets up the computational graph for the forward computation. To actually run the network we need to enter a TensorFlow Session and feed data to the computational graph.

After defining the network architecture in the `two_layer_fc` function, we will test the implementation by setting up and running a computational graph, feeding zeros to the network and checking the shape of the output.

It's important that you read and understand this implementation.

```
In [13]: def two_layer_fc(x, params):
             """
             A fully-connected neural network; the architecture is:
             fully-connected layer -> ReLU -> fully connected layer.
             Note that we only need to define the forward pass here; TensorFlow will take
             care of computing the gradients for us.

             The input to the network will be a minibatch of data, of shape
             (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H units,
             and the output layer will produce scores for C classes.

             Inputs:
             - x: A TensorFlow Tensor of shape (N, d1, ..., dM) giving a minibatch of
               input data.
             - params: A list [w1, w2] of TensorFlow Tensors giving weights for the
               network, where w1 has shape (D, H) and w2 has shape (H, C).

             Returns:
             - scores: A TensorFlow Tensor of shape (N, C) giving classification scores
               for the input data x.
```

```
    """
    w1, w2 = params  # Unpack the parameters
    x = flatten(x)   # Flatten the input; now x has shape (N, D)
    h = tf.nn.relu(tf.matmul(x, w1)) # Hidden layer: h has shape (N, H)
    scores = tf.matmul(h, w2)        # Compute scores of shape (N, C)
    return scores
```

```
In [14]: def two_layer_fc_test():
             # TensorFlow's default computational graph is essentially a hidden global
             # variable. To avoid adding to this default graph when you rerun this cell,
             # we clear the default graph before constructing the graph we care about.
             tf.reset_default_graph()
             hidden_layer_size = 42

             # Scoping our computational graph setup code under a tf.device context
             # manager lets us tell TensorFlow where we want these Tensors to be
             # placed.
             with tf.device(device):
                 # Set up a placehoder for the input of the network, and constant
                 # zero Tensors for the network weights. Here we declare w1 and w2
                 # using tf.zeros instead of tf.placeholder as we've seen before - this
                 # means that the values of w1 and w2 will be stored in the computational
                 # graph itself and will persist across multiple runs of the graph; in
                 # particular this means that we don't have to pass values for w1 and w2
                 # using a feed_dict when we eventually run the graph.
                 x = tf.placeholder(tf.float32)
                 w1 = tf.zeros((32 * 32 * 3, hidden_layer_size))
                 w2 = tf.zeros((hidden_layer_size, 10))

                 # Call our two_layer_fc function to set up the computational
                 # graph for the forward pass of the network.
                 scores = two_layer_fc(x, [w1, w2])

             # Use numpy to create some concrete data that we will pass to the
             # computational graph for the x placeholder.
             x_np = np.zeros((64, 32, 32, 3))
             with tf.Session() as sess:
                 # The calls to tf.zeros above do not actually instantiate the values
                 # for w1 and w2; the following line tells TensorFlow to instantiate
                 # the values of all Tensors (like w1 and w2) that live in the graph.
                 sess.run(tf.global_variables_initializer())

                 # Here we actually run the graph, using the feed_dict to pass the
                 # value to bind to the placeholder for x; we ask TensorFlow to compute
                 # the value of the scores Tensor, which it returns as a numpy array.
                 scores_np = sess.run(scores, feed_dict={x: x_np})
                 print(scores_np.shape)
```

```
         two_layer_fc_test()
```

```
(64, 10)
```

4.0.3 Barebones TensorFlow: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet` which will perform the forward pass of a three-layer convolutional network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for `C` classes.

HINT: For convolutions: https://www.tensorflow.org/api_docs/python/tf/nn/conv2d; be careful with padding!

HINT: For biases: https://www.tensorflow.org/performance/xla/broadcasting

```
In [23]: def three_layer_convnet(x, params):
             """
             A three-layer convolutional network with the architecture described above.

             Inputs:
             - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of images
             - params: A list of TensorFlow Tensors giving the weights and biases for the
               network; should contain the following:
               - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
                 weights for the first convolutional layer.
               - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
                 first convolutional layer.
               - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
                 giving weights for the second convolutional layer
               - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
                 second convolutional layer.
               - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
                 Can you figure out what the shape should be?
               - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
                 Can you figure out what the shape should be?
             """
             conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
             scores = None
             ############################################################################
             # TODO: Implement the forward pass for the three-layer ConvNet.            #
             ############################################################################
```

```
    pass
    #############################################################################
    #                               END OF YOUR CODE                            #
    #############################################################################
    return scores
```

After defing the forward pass of the three-layer ConvNet above, run the following cell to test your implementation. Like the two-layer network, we use the `three_layer_convnet` function to set up the computational graph, then run the graph on a batch of zeros just to make sure the function doesn't crash, and produces outputs of the correct shape.

When you run this function, `scores_np` should have shape `(64, 10)`.

```
In [ ]: def three_layer_convnet_test():
            tf.reset_default_graph()

            with tf.device(device):
                x = tf.placeholder(tf.float32)
                conv_w1 = tf.zeros((5, 5, 3, 6))
                conv_b1 = tf.zeros((6,))
                conv_w2 = tf.zeros((3, 3, 6, 9))
                conv_b2 = tf.zeros((9,))
                fc_w = tf.zeros((32 * 32 * 9, 10))
                fc_b = tf.zeros((10,))
                params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
                scores = three_layer_convnet(x, params)

            # Inputs to convolutional layers are 4-dimensional arrays with shape
            # [batch_size, height, width, channels]
            x_np = np.zeros((64, 32, 32, 3))

            with tf.Session() as sess:
                sess.run(tf.global_variables_initializer())
                scores_np = sess.run(scores, feed_dict={x: x_np})
                print('scores_np has shape: ', scores_np.shape)

        with tf.device('/cpu:0'):
            three_layer_convnet_test()
```

4.0.4 Barebones TensorFlow: Training Step

We now define the `training_step` function which sets up the part of the computational graph that performs a single training step. This will take three basic steps:

1. Compute the loss
2. Compute the gradient of the loss with respect to all network weights
3. Make a weight update step using (stochastic) gradient descent.

Note that the step of updating the weights is itself an operation in the computational graph - the calls to `tf.assign_sub` in `training_step` return TensorFlow operations that mutate the weights when they are executed. There is an important bit of subtlety here - when we call sess.run, TensorFlow does not execute all operations in the computational graph; it only executes the minimal subset of the graph necessary to compute the outputs that we ask TensorFlow to produce. As a result, naively computing the loss would not cause the weight update operations to execute, since the operations needed to compute the loss do not depend on the output of the weight update. To fix this problem, we insert a **control dependency** into the graph, adding a duplicate loss node to the graph that does depend on the outputs of the weight update operations; this is the object that we actually return from the training_step function. As a result, asking TensorFlow to evaluate the value of the loss returned from training_step will also implicitly update the weights of the network using that minibatch of data.

We need to use a few new TensorFlow functions to do all of this: - For computing the cross-entropy loss we'll use tf.nn.sparse_softmax_cross_entropy_with_logits: https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits - For averaging the loss across a minibatch of data we'll use tf.reduce_mean: https://www.tensorflow.org/api_docs/python/tf/reduce_mean - For computing gradients of the loss with respect to the weights we'll use tf.gradients: https://www.tensorflow.org/api_docs/python/tf/gradients - We'll mutate the weight values stored in a TensorFlow Tensor using tf.assign_sub: https://www.tensorflow.org/api_docs/python/tf/assign_sub - We'll add a control dependency to the graph using tf.control_dependencies: https://www.tensorflow.org/api_docs/python/tf/control_dependencies

```
In [ ]: def training_step(scores, y, params, learning_rate):
            """
            Set up the part of the computational graph which makes a training step.

            Inputs:
            - scores: TensorFlow Tensor of shape (N, C) giving classification scores for
              the model.
            - y: TensorFlow Tensor of shape (N,) giving ground-truth labels for scores;
              y[i] == c means that c is the correct class for scores[i].
            - params: List of TensorFlow Tensors giving the weights of the model
            - learning_rate: Python scalar giving the learning rate to use for gradient
              descent step.

            Returns:
            - loss: A TensorFlow Tensor of shape () (scalar) giving the loss for this
              batch of data; evaluating the loss also performs a gradient descent step
              on params (see above).
            """
            # First compute the loss; the first line gives losses for each example in
            # the minibatch, and the second averages the losses acros the batch
            losses = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=scores)
            loss = tf.reduce_mean(losses)

            # Compute the gradient of the loss with respect to each parameter of the the
            # network. This is a very magical function call: TensorFlow internally
            # traverses the computational graph starting at loss backward to each element
```

```
    # of params, and uses backpropagation to figure out how to compute gradients;
    # it then adds new operations to the computational graph which compute the
    # requested gradients, and returns a list of TensorFlow Tensors that will
    # contain the requested gradients when evaluated.
    grad_params = tf.gradients(loss, params)

    # Make a gradient descent step on all of the model parameters.
    new_weights = []
    for w, grad_w in zip(params, grad_params):
        new_w = tf.assign_sub(w, learning_rate * grad_w)
        new_weights.append(new_w)

    # Insert a control dependency so that evaluting the loss causes a weight
    # update to happen; see the discussion above.
    with tf.control_dependencies(new_weights):
        return tf.identity(loss)
```

4.0.5 Barebones TensorFlow: Training Loop

Now we set up a basic training loop using low-level TensorFlow operations. We will train the model using stochastic gradient descent without momentum. The `training_step` function sets up the part of the computational graph that performs the training step, and the function `train_part2` iterates through the training data, making training steps on each minibatch, and periodically evaluates accuracy on the validation set.

```
In [ ]: def train_part2(model_fn, init_fn, learning_rate):
            """
            Train a model on CIFAR-10.

            Inputs:
            - model_fn: A Python function that performs the forward pass of the model
              using TensorFlow; it should have the following signature:
              scores = model_fn(x, params) where x is a TensorFlow Tensor giving a
              minibatch of image data, params is a list of TensorFlow Tensors holding
              the model weights, and scores is a TensorFlow Tensor of shape (N, C)
              giving scores for all elements of x.
            - init_fn: A Python function that initializes the parameters of the model.
              It should have the signature params = init_fn() where params is a list
              of TensorFlow Tensors holding the (randomly initialized) weights of the
              model.
            - learning_rate: Python float giving the learning rate to use for SGD.
            """
            # First clear the default graph
            tf.reset_default_graph()
            is_training = tf.placeholder(tf.bool, name='is_training')
            # Set up the computational graph for performing forward and backward passes,
            # and weight updates.
            with tf.device(device):
                # Set up placeholders for the data and labels
                x = tf.placeholder(tf.float32, [None, 32, 32, 3])
                y = tf.placeholder(tf.int32, [None])
                params = init_fn()            # Initialize the model parameters
                scores = model_fn(x, params) # Forward pass of the model
                loss = training_step(scores, y, params, learning_rate)

            # Now we actually run the graph many times using the training data
            with tf.Session() as sess:
                # Initialize variables that will live in the graph
                sess.run(tf.global_variables_initializer())
                for t, (x_np, y_np) in enumerate(train_dset):
                    # Run the graph on a batch of training data; recall that asking
                    # TensorFlow to evaluate loss will cause an SGD step to happen.
                    feed_dict = {x: x_np, y: y_np}
                    loss_np = sess.run(loss, feed_dict=feed_dict)

                    # Periodically print the loss and check accuracy on the val set
                    if t % print_every == 0:
                        print('Iteration %d, loss = %.4f' % (t, loss_np))
                        check_accuracy(sess, val_dset, x, scores, is_training)
```

4.0.6 Barebones TensorFlow: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets. Note that this function accepts a TensorFlow Session object as one of its arguments; this is needed since the function must actually run the computational graph many times on the data that it loads from the dataset `dset`.

Also note that we reuse the same computational graph both for taking training steps and for evaluating the model; however since the `check_accuracy` function never evalutes the `loss` value in the computational graph, the part of the graph that updates the weights of the graph do not execute on the validation data.

```
In [ ]: def check_accuracy(sess, dset, x, scores, is_training=None):
            """
            Check accuracy on a classification model.

            Inputs:
            - sess: A TensorFlow Session that will be used to run the graph
            - dset: A Dataset object on which to check accuracy
            - x: A TensorFlow placeholder Tensor where input images should be fed
            - scores: A TensorFlow Tensor representing the scores output from the
              model; this is the Tensor we will ask TensorFlow to evaluate.

            Returns: Nothing, but prints the accuracy of the model
            """
            num_correct, num_samples = 0, 0
            for x_batch, y_batch in dset:
                feed_dict = {x: x_batch, is_training: 0}
                scores_np = sess.run(scores, feed_dict=feed_dict)
                y_pred = scores_np.argmax(axis=1)
                num_samples += x_batch.shape[0]
                num_correct += (y_pred == y_batch).sum()
            acc = float(num_correct) / num_samples
            print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * acc))
```

4.0.7 Barebones TensorFlow: Initialization

We'll use the following utility method to initialize the weight matrices for our models using Kaiming's normalization method.

[1] He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* , ICCV 2015, https://arxiv.org/abs/1502.01852

```
In [ ]: def kaiming_normal(shape):
            if len(shape) == 2:
                fan_in, fan_out = shape[0], shape[1]
            elif len(shape) == 4:
                fan_in, fan_out = np.prod(shape[:3]), shape[3]
            return tf.random_normal(shape) * np.sqrt(2.0 / fan_in)
```

4.0.8 Barebones TensorFlow: Train a Two-Layer Network

We are finally ready to use all of the pieces defined above to train a two-layer fully-connected network on CIFAR-10.

We just need to define a function to initialize the weights of the model, and call `train_part2`.

Defining the weights of the network introduces another important piece of TensorFlow API: `tf.Variable`. A TensorFlow Variable is a Tensor whose value is stored in the graph and persists across runs of the computational graph; however unlike constants defined with `tf.zeros` or `tf.random_normal`, the values of a Variable can be mutated as the graph runs; these mutations will persist across graph runs. Learnable parameters of the network are usually stored in Variables.

You don't need to tune any hyperparameters, but you should achieve accuracies above 40% after one epoch of training.

```
In [ ]: def two_layer_fc_init():
            """
            Initialize the weights of a two-layer network, for use with the
            two_layer_network function defined above.

            Inputs: None

            Returns: A list of:
            - w1: TensorFlow Variable giving the weights for the first layer
            - w2: TensorFlow Variable giving the weights for the second layer
            """
            hidden_layer_size = 4000
            w1 = tf.Variable(kaiming_normal((3 * 32 * 32, 4000)))
            w2 = tf.Variable(kaiming_normal((4000, 10)))
```

```
    return [w1, w2]

learning_rate = 1e-2
train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

4.0.9 Barebones TensorFlow: Train a three-layer ConvNet

We will now use TensorFlow to train a three-layer ConvNet on CIFAR-10.

You need to implement the `three_layer_convnet_init` function. Recall that the architecture of the network is:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You don't need to do any hyperparameter tuning, but you should see accuracies above 43% after one epoch of training.

```
In [ ]: def three_layer_convnet_init():
            """
            Initialize the weights of a Three-Layer ConvNet, for use with the
            three_layer_convnet function defined above.

            Inputs: None

            Returns a list containing:
            - conv_w1: TensorFlow Variable giving weights for the first conv layer
            - conv_b1: TensorFlow Variable giving biases for the first conv layer
            - conv_w2: TensorFlow Variable giving weights for the second conv layer
            - conv_b2: TensorFlow Variable giving biases for the second conv layer
            - fc_w: TensorFlow Variable giving weights for the fully-connected layer
            - fc_b: TensorFlow Variable giving biases for the fully-connected layer
            """
            params = None
            ############################################################################
            # TODO: Initialize the parameters of the three-layer network.              #
            ############################################################################
            pass
            ############################################################################
            #                             END OF YOUR CODE                             #
            ############################################################################
            return params

        learning_rate = 3e-3
        train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

5 Part III: Keras Model API

Implementing a neural network using the low-level TensorFlow API is a good way to understand how TensorFlow works, but it's a little inconvenient - we had to manually keep track of all Tensors holding learnable parameters, and we had to use a control dependency to implement the gradient descent update step. This was fine for a small network, but could quickly become unweildy for a large complex model.

Fortunately TensorFlow provides higher-level packages such as `tf.keras` and `tf.layers` which make it easy to build models out of modular, object-oriented layers; `tf.train` allows you to easily train these models using a variety of different optimization algorithms.

In this part of the notebook we will define neural network models using the `tf.keras.Model` API. To implement your own model, you need to do the following:

1. Define a new class which subclasses `tf.keras.model`. Give your class an intuitive name that describes it, like `TwoLayerFC` or `ThreeLayerConvNet`.
2. In the initializer `__init__()` for your new class, define all the layers you need as class attributes. The `tf.layers` package provides many common neural-network layers, like `tf.layers.Dense` for fully-connected layers and `tf.layers.Conv2D` for convolutional layers. Under the hood, these layers will construct `Variable` Tensors for any learnable parameters. **Warning**: Don't forget to call `super().__init__()` as the first line in your initializer!
3. Implement the `call()` method for your class; this implements the forward pass of your model, and defines the *connectivity* of your network. Layers defined in `__init__()` implement `__call__()` so they can be used as function objects that transform input Tensors into output Tensors. Don't define any new layers in `call()`; any layers you want to use in the forward pass should be defined in `__init__()`.

After you define your `tf.keras.Model` subclass, you can instantiate it and use it like the model functions from Part II.

5.0.1 Module API: Two-Layer Network

Here is a concrete example of using the `tf.keras.Model` API to define a two-layer network. There are a few new bits of API to be aware of here:

We use an `Initializer` object to set up the initial values of the learnable parameters of the layers; in particular `tf.variance_scaling_initializer` gives behavior similar to the Kaiming initialization method we used in Part II. You can read more about it here: https://www.tensorflow.org/api_docs/python/tf/variance_scaling_initializer

We construct `tf.layers.Dense` objects to represent the two fully-connected layers of the model. In addition to multiplying their input by a weight matrix and adding a bias vector, these layer can also apply a nonlinearity for you. For the first layer we specify a ReLU activation function by passing `activation=tf.nn.relu` to the constructor; the second layer does not apply any activation function.

Unfortunately the `flatten` function we defined in Part II is not compatible with the `tf.keras.Model` API; fortunately we can use `tf.layers.flatten` to perform the same operation. The issue with our `flatten` function from Part II has to do with static vs dynamic shapes for Tensors, which is beyond the scope of this notebook; you can read more about the distinction in the documentation.

```
In [25]: class TwoLayerFC(tf.keras.Model):
             def __init__(self, hidden_size, num_classes):
                 super().__init__()
                 initializer = tf.variance_scaling_initializer(scale=2.0)
                 self.fc1 = tf.layers.Dense(hidden_size, activation=tf.nn.relu,
                                            kernel_initializer=initializer)
                 self.fc2 = tf.layers.Dense(num_classes,
                                            kernel_initializer=initializer)
             def call(self, x, training=None):
                 x = tf.layers.flatten(x)
                 x = self.fc1(x)
                 x = self.fc2(x)
                 return x


         def test_TwoLayerFC():
             """ A small unit test to exercise the TwoLayerFC model above. """
             tf.reset_default_graph()
             input_size, hidden_size, num_classes = 50, 42, 10

             # As usual in TensorFlow, we first need to define our computational graph.
             # To this end we first construct a TwoLayerFC object, then use it to construct
             # the scores Tensor.
             model = TwoLayerFC(hidden_size, num_classes)
             with tf.device(device):
                 x = tf.zeros((64, input_size))
                 scores = model(x)

             # Now that our computational graph has been defined we can run the graph
             with tf.Session() as sess:
                 sess.run(tf.global_variables_initializer())
                 scores_np = sess.run(scores)
                 print(scores_np.shape)

         test_TwoLayerFC()
```

```
(64, 10)
```

5.0.2 Funtional API: Two-Layer Network

The `tf.layers` package provides two different higher-level APIs for defining neural network models. In the example above we used the **object-oriented API**, where each layer of the neural network is represented as a Python object (like `tf.layers.Dense`). Here we showcase the **functional API**, where each layer is a Python function (like `tf.layers.dense`) which inputs and outputs TensorFlow Tensors, and which internally sets up Tensors in the computational graph to hold any learnable weights.

To construct a network, one needs to pass the input tensor to the first layer, and construct the subsequent layers sequentially. Here's an example of how to construct the same two-layer nework with the functional API.

```
In [26]: def two_layer_fc_functional(inputs, hidden_size, num_classes):
             initializer = tf.variance_scaling_initializer(scale=2.0)
             flattened_inputs = tf.layers.flatten(inputs)
             fc1_output = tf.layers.dense(flattened_inputs, hidden_size, activation=tf.nn.relu
                                          kernel_initializer=initializer)
             scores = tf.layers.dense(fc1_output, num_classes,
                                      kernel_initializer=initializer)
             return scores

         def test_two_layer_fc_functional():
             """ A small unit test to exercise the TwoLayerFC model above. """
             tf.reset_default_graph()
             input_size, hidden_size, num_classes = 50, 42, 10

             # As usual in TensorFlow, we first need to define our computational graph.
             # To this end we first construct a two layer network graph by calling the
             # two_layer_network() function. This function constructs the computation
             # graph and outputs the score tensor.
             with tf.device(device):
                 x = tf.zeros((64, input_size))
                 scores = two_layer_fc_functional(x, hidden_size, num_classes)

             # Now that our computational graph has been defined we can run the graph
             with tf.Session() as sess:
                 sess.run(tf.global_variables_initializer())
                 scores_np = sess.run(scores)
                 print(scores_np.shape)

         test_two_layer_fc_functional()
```

```
(64, 10)
```

5.0.3 Keras Model API: Three-Layer ConvNet

Now it's your turn to implement a three-layer ConvNet using the `tf.keras.Model` API. Your model should have the same architecture used in Part II:

1. Convolutional layer with 5 x 5 kernels, with zero-padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 3 x 3 kernels, with zero-padding of 1
4. ReLU nonlinearity
5. Fully-connected layer to give class scores

You should initialize the weights of your network using the same initialization method as was used in the two-layer network above.

Hint: Refer to the documentation for `tf.layers.Conv2D` and `tf.layers.Dense`:
https://www.tensorflow.org/api_docs/python/tf/layers/Conv2D
https://www.tensorflow.org/api_docs/python/tf/layers/Dense

```
In [27]: class ThreeLayerConvNet(tf.keras.Model):
             def __init__(self, channel_1, channel_2, num_classes):
                 super().__init__()
                 #########################################################################
                 # TODO: Implement the __init__ method for a three-layer ConvNet. You    #
                 # should instantiate layer objects to be used in the forward pass.      #
                 #########################################################################
                 pass
                 #########################################################################
                 #                            END OF YOUR CODE                            #
                 #########################################################################

             def call(self, x, training=None):
                 scores = None
                 #########################################################################
                 # TODO: Implement the forward pass for a three-layer ConvNet. You       #
                 # should use the layer objects defined in the __init__ method.          #
                 #########################################################################
                 pass
                 #########################################################################
                 #                            END OF YOUR CODE                            #
                 #########################################################################
                 return scores
```

Once you complete the implementation of the `ThreeLayerConvNet` above you can run the following to ensure that your implementation does not crash and produces outputs of the expected shape.

```
In [ ]: def test_ThreeLayerConvNet():
            tf.reset_default_graph()

            channel_1, channel_2, num_classes = 12, 8, 10
            model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
            with tf.device(device):
                x = tf.zeros((64, 3, 32, 32))
                scores = model(x)

            with tf.Session() as sess:
                sess.run(tf.global_variables_initializer())
                scores_np = sess.run(scores)
                print(scores_np.shape)

        test_ThreeLayerConvNet()
```

5.0.4 Keras Model API: Training Loop

We need to implement a slightly different training loop when using the tf.keras.Model API. Instead of computing gradients and updating the weights of the model manually, we use an Optimizer object from the tf.train package which takes care of these details for us. You can read more about Optimizers here: https://www.tensorflow.org/api_docs/python/tf/train/Optimizer

```
In [29]: def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1):
             """
             Simple training loop for use with models defined using tf.keras. It trains
             a model for one epoch on the CIFAR-10 training set and periodically checks
             accuracy on the CIFAR-10 validation set.

             Inputs:
             - model_init_fn: A function that takes no parameters; when called it
               constructs the model we want to train: model = model_init_fn()
             - optimizer_init_fn: A function which takes no parameters; when called it
               constructs the Optimizer object we will use to optimize the model:
               optimizer = optimizer_init_fn()
             - num_epochs: The number of epochs to train for

             Returns: Nothing, but prints progress during trainingn
             """
             tf.reset_default_graph()
             with tf.device(device):
                 # Construct the computational graph we will use to train the model. We
                 # use the model_init_fn to construct the model, declare placeholders for
                 # the data and labels
                 x = tf.placeholder(tf.float32, [None, 32, 32, 3])
                 y = tf.placeholder(tf.int32, [None])

                 # We need a place holder to explicitly specify if the model is in the trainin
                 # phase or not. This is because a number of layers behaves differently in
                 # training and in testing, e.g., dropout and batch normalization.
                 # We pass this variable to the computation graph through feed_dict as shown b
                 is_training = tf.placeholder(tf.bool, name='is_training')

                 # Use the model function to build the forward pass.
                 scores = model_init_fn(x, is_training)

                 # Compute the loss like we did in Part II
                 loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=scores
                 loss = tf.reduce_mean(loss)

                 # Use the optimizer_fn to construct an Optimizer, then use the optimizer
                 # to set up the training step. Asking TensorFlow to evaluate the
                 # train_op returned by optimizer.minimize(loss) will cause us to make a
                 # single update step using the current minibatch of data.
```

```
        # Note that we use tf.control_dependencies to force the model to run
        # the tf.GraphKeys.UPDATE_OPS at each training step. tf.GraphKeys.UPDATE_OPS
        # holds the operators that update the states of the network.
        # For example, the tf.layers.batch_normalization function adds the running me
        # and variance update operators to tf.GraphKeys.UPDATE_OPS.
        optimizer = optimizer_init_fn()
        update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
        with tf.control_dependencies(update_ops):
            train_op = optimizer.minimize(loss)

    # Now we can run the computational graph many times to train the model.
    # When we call sess.run we ask it to evaluate train_op, which causes the
    # model to update.
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        t = 0
        for epoch in range(num_epochs):
            print('Starting epoch %d' % epoch)
            for x_np, y_np in train_dset:
                feed_dict = {x: x_np, y: y_np, is_training:1}
                loss_np, _ = sess.run([loss, train_op], feed_dict=feed_dict)
                if t % print_every == 0:
                    print('Iteration %d, loss = %.4f' % (t, loss_np))
                    check_accuracy(sess, val_dset, x, scores, is_training=is_training
                    print()
                t += 1
```

5.0.5 Keras Model API: Train a Two-Layer Network

We can now use the tools defined above to train a two-layer network on CIFAR-10. We define the model_init_fn and optimizer_init_fn that construct the model and optimizer respectively when called. Here we want to train the model using stochastic gradient descent with no momentum, so we construct a tf.train.GradientDescentOptimizer function; you can read about it here.

You don't need to tune any hyperparameters here, but you should achieve accuracies above 40% after one epoch of training.

```
In [ ]: hidden_size, num_classes = 4000, 10
        learning_rate = 1e-2

        def model_init_fn(inputs, is_training):
            return TwoLayerFC(hidden_size, num_classes)(inputs)

        def optimizer_init_fn():
            return tf.train.GradientDescentOptimizer(learning_rate)

        train_part34(model_init_fn, optimizer_init_fn)
```

5.0.6 Keras Model API: Train a Two-Layer Network (functional API)

Similarly, we train the two-layer network constructed using the functional API.

```
In [ ]: hidden_size, num_classes = 4000, 10
        learning_rate = 1e-2

        def model_init_fn(inputs, is_training):
            return two_layer_fc_functional(inputs, hidden_size, num_classes)

        def optimizer_init_fn():
            return tf.train.GradientDescentOptimizer(learning_rate)

        train_part34(model_init_fn, optimizer_init_fn)
```

5.0.7 Keras Model API: Train a Three-Layer ConvNet

Here you should use the tools we've defined above to train a three-layer ConvNet on CIFAR-10. Your ConvNet should use 32 filters in the first convolutional layer and 16 filters in the second layer.

To train the model you should use gradient descent with Nesterov momentum 0.9.

HINT: https://www.tensorflow.org/api_docs/python/tf/train/MomentumOptimizer

You don't need to perform any hyperparameter tuning, but you should achieve accuracies above 45% after training for one epoch.

```
In [ ]: learning_rate = 3e-3
        channel_1, channel_2, num_classes = 32, 16, 10

        def model_init_fn(inputs, is_training):
            model = None
            ############################################################################
            # TODO: Complete the implementation of model_fn.                           #
            ############################################################################
            ############################################################################
            #                           END OF YOUR CODE                               #
            ############################################################################
            return model(inputs)

        def optimizer_init_fn():
            optimizer = None
            ############################################################################
            # TODO: Complete the implementation of model_fn.                           #
            ############################################################################
            ############################################################################
            #                           END OF YOUR CODE                               #
            ############################################################################
            return optimizer

        train_part34(model_init_fn, optimizer_init_fn)
```

6 Part IV: Keras Sequential API

In Part III we introduced the `tf.keras.Model` API, which allows you to define models with any number of learnable layers and with arbitrary connectivity between layers.

However for many models you don't need such flexibility - a lot of models can be expressed as a sequential stack of layers, with the output of each layer fed to the next layer as input. If your model fits this pattern, then there is an even easier way to define your model: using `tf.keras.Sequential`. You don't need to write any custom classes; you simply call the `tf.keras.Sequential` constructor with a list containing a sequence of layer objects.

One complication with `tf.keras.Sequential` is that you must define the shape of the input to the model by passing a value to the `input_shape` of the first layer in your model.

6.0.1 Keras Sequential API: Two-Layer Network

Here we rewrite the two-layer fully-connected network using `tf.keras.Sequential`, and train it using the training loop defined above.

You don't need to perform any hyperparameter tuning here, but you should see accuracies above 40% after training for one epoch.

```
In [ ]: learning_rate = 1e-2

        def model_init_fn(inputs, is_training):
            input_shape = (32, 32, 3)
            hidden_layer_size, num_classes = 4000, 10
            initializer = tf.variance_scaling_initializer(scale=2.0)
            layers = [
                tf.layers.Flatten(input_shape=input_shape),
                tf.layers.Dense(hidden_layer_size, activation=tf.nn.relu,
                                kernel_initializer=initializer),
                tf.layers.Dense(num_classes, kernel_initializer=initializer),
            ]
            model = tf.keras.Sequential(layers)
            return model(inputs)

        def optimizer_init_fn():
            return tf.train.GradientDescentOptimizer(learning_rate)

        train_part34(model_init_fn, optimizer_init_fn)
```

6.0.2 Keras Sequential API: Three-Layer ConvNet

Here you should use `tf.keras.Sequential` to reimplement the same three-layer ConvNet architecture used in Part II and Part III. As a reminder, your model should have the following architecture:

1. Convolutional layer with 16 5x5 kernels, using zero padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 32 3x3 kernels, using zero padding of 1
4. ReLU nonlinearity

5. Fully-connected layer giving class scores

You should initialize the weights of the model using a `tf.variance_scaling_initializer` as above.

You should train the model using Nesterov momentum 0.9.

You don't need to perform any hyperparameter search, but you should achieve accuracy above 45% after training for one epoch.

```
In [ ]: def model_init_fn(inputs, is_training):
            model = None
            ############################################################################
            # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.         #
            ############################################################################
            pass
            ############################################################################
            #                             END OF YOUR CODE                             #
            ############################################################################
            return model(inputs)

        learning_rate = 5e-4
        def optimizer_init_fn():
            optimizer = None
            ############################################################################
            # TODO: Complete the implementation of model_fn.                           #
            ############################################################################
            ############################################################################
            #                            END OF YOUR CODE                              #
            ############################################################################
            return optimizer

        train_part34(model_init_fn, optimizer_init_fn)
```

7 Part V: CIFAR-10 open-ended challenge

In this section you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

You should experiment with architectures, hyperparameters, loss functions, regularization, or anything else you can think of to train a model that achieves **at least 70%** accuracy on the **validation** set within 10 epochs. You can use the `check_accuracy` and `train` functions from above, or you can implement your own training loop.

Describe what you did at the end of the notebook.

7.0.1 Some things you can try:

- **Filter size**: Above we used 5x5 and 3x3; is this optimal?
- **Number of filters**: Above we used 16 and 32 filters. Would more or fewer do better?
- **Pooling**: We didn't use any pooling above. Would this improve the model?
- **Normalization**: Would your model be improved with batch normalization, layer normalization, group normalization, or some other normalization strategy?

- **Network architecture**: The ConvNet above has only three layers of trainable parameters. Would a deeper model do better?
- **Global average pooling**: Instead of flattening after the final convolutional layer, would global average pooling do better? This strategy is used for example in Google's Inception network and in Residual Networks.
- **Regularization**: Would some kind of regularization improve performance? Maybe weight decay or dropout?

7.0.2 WARNING: Batch Normalization / Dropout

Batch Normalization and Dropout **WILL NOT WORK CORRECTLY** if you use the `train_part34()` function with the object-oriented `tf.keras.Model` or `tf.keras.Sequential` APIs; if you want to use these layers with this training loop then you **must use the tf.layers functional API**.

We wrote `train_part34()` to explicitly demonstrate how TensorFlow works; however there are some subtleties that make it tough to handle the object-oriented batch normalization layer in a simple training loop. In practice both `tf.keras` and `tf` provide higher-level APIs which handle the training loop for you, such as keras.fit and tf.Estimator, both of which will properly handle batch normalization when using the object-oriented API.

7.0.3 Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

7.0.4 Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
- ResNets where the input from the previous layer is added to the output.
- DenseNets where inputs into previous layers are concatenated together.
- This blog has an in-depth overview

7.0.5 Have fun and happy training!

```
In [ ]: def model_init_fn(inputs, is_training):
            model = None
            ############################################################################
            # TODO: Construct a model that performs well on CIFAR-10                   #
            ############################################################################
            pass
            ############################################################################
            #                            END OF YOUR CODE                              #
            ############################################################################
            return net

        pass

        def optimizer_init_fn():
            optimizer = None
            ############################################################################
            # TODO: Construct an optimizer that performs well on CIFAR-10              #
            ############################################################################
            pass
            ############################################################################
            #                            END OF YOUR CODE                              #
            ############################################################################
            return optimizer

        device = '/gpu:0'
        print_every = 700
        num_epochs = 10
        train_part34(model_init_fn, optimizer_init_fn, num_epochs)
```

7.1 Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

TODO: Tell us what you did