

Fully-Connected Neural Nets

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

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

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

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

    return out, cache
```

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

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

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

    return dx, dw
```

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

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

```
In [1]: # As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

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

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

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

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

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

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

Affine layer: forward

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

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

```
In [40]: # Test the affine_forward function

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

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

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

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

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

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

Affine layer: backward

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

```
In [41]: # Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

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

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

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

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

ReLU activation: forward

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

```
In [42]: # Test the relu_forward function

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

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,
                           ],
                        [ 0.,          0.,          0.04545455,  0.13636
364, ],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,
                           ]])

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

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

ReLU activation: backward

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

```
In [43]: 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, d
out)

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

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

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

Inline Question 1:

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

1. Sigmoid
2. ReLU
3. Leaky ReLU

Answer:

1. Sigmoid would have zero gradient (gradient loss), because of the function is a "S-shape" curve, when the input x is too large or too small, the gradient is close to zero.
2. ReLU, when the input x is greater than zero, the gradient is one which is good, but when the input x is less than zero, the gradient becomes zero (gradient loss for the negative input)
3. Leaky ReLU, this would be the best activation (in my mind) because it keep the good part of ReLU which has gradient equal to one when the input is positive, for the negative input, it gives 0.001 (hyper-parameter) gradient which won't cause gradient loss

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

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

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

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

Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax and SVM

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

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

```
In [45]: np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

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

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

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

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

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

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

Two-layer network

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

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


```

In [46]: np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

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

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

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

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

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

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

    for name in sorted(grads):

```

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

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.83e-08
W2 relative error: 3.31e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

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

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

```

In [37]: model = TwoLayerNet()
solver = None

#####
#####
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least
# 50% accuracy on the validation set.
#
#####
#####
data = {
    'X_train': data['X_train'], # training data
    'y_train': data['y_train'], # training labels
    'X_val': data['X_val'], # validation data
    'y_val': data['y_val'] # validation labels
}
solver = Solver(model,
                 data,
                 update_rule='sgd',
                 optim_config={'learning_rate': 1e-3},
                 lr_decay=0.95,
                 num_epochs=10,
                 batch_size=100,
                 print_every=100)
solver.train()
#####
#####
#
#                                     END OF YOUR CODE
#
#####
#####

```

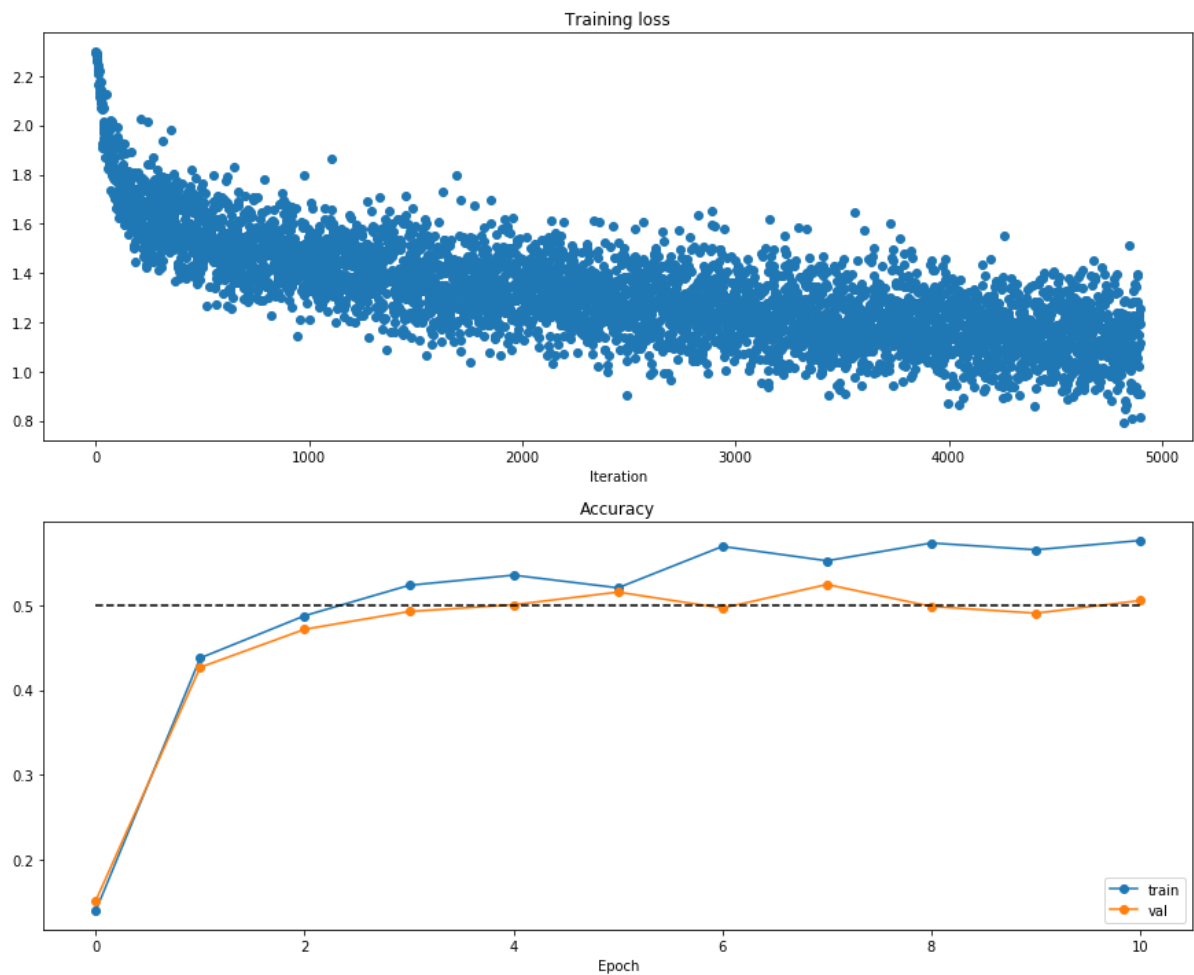
```
(Iteration 1 / 4900) loss: 2.300161
(Epoch 0 / 10) train acc: 0.139000; val_acc: 0.151000
(Iteration 101 / 4900) loss: 1.777724
(Iteration 201 / 4900) loss: 1.681457
(Iteration 301 / 4900) loss: 1.515172
(Iteration 401 / 4900) loss: 1.707817
(Epoch 1 / 10) train acc: 0.438000; val_acc: 0.427000
(Iteration 501 / 4900) loss: 1.765811
(Iteration 601 / 4900) loss: 1.516779
(Iteration 701 / 4900) loss: 1.714298
(Iteration 801 / 4900) loss: 1.540492
(Iteration 901 / 4900) loss: 1.572720
(Epoch 2 / 10) train acc: 0.488000; val_acc: 0.472000
(Iteration 1001 / 4900) loss: 1.472802
(Iteration 1101 / 4900) loss: 1.419539
(Iteration 1201 / 4900) loss: 1.335037
(Iteration 1301 / 4900) loss: 1.589125
(Iteration 1401 / 4900) loss: 1.492380
(Epoch 3 / 10) train acc: 0.524000; val_acc: 0.493000
(Iteration 1501 / 4900) loss: 1.241189
(Iteration 1601 / 4900) loss: 1.203644
(Iteration 1701 / 4900) loss: 1.316281
(Iteration 1801 / 4900) loss: 1.429428
(Iteration 1901 / 4900) loss: 1.066646
(Epoch 4 / 10) train acc: 0.536000; val_acc: 0.501000
(Iteration 2001 / 4900) loss: 1.459705
(Iteration 2101 / 4900) loss: 1.310179
(Iteration 2201 / 4900) loss: 1.418849
(Iteration 2301 / 4900) loss: 1.344758
(Iteration 2401 / 4900) loss: 1.260249
(Epoch 5 / 10) train acc: 0.521000; val_acc: 0.516000
(Iteration 2501 / 4900) loss: 1.141024
(Iteration 2601 / 4900) loss: 1.314819
(Iteration 2701 / 4900) loss: 1.153019
(Iteration 2801 / 4900) loss: 1.221160
(Iteration 2901 / 4900) loss: 1.349037
(Epoch 6 / 10) train acc: 0.570000; val_acc: 0.497000
(Iteration 3001 / 4900) loss: 1.313611
(Iteration 3101 / 4900) loss: 1.168867
(Iteration 3201 / 4900) loss: 1.028615
(Iteration 3301 / 4900) loss: 1.225758
(Iteration 3401 / 4900) loss: 1.333077
(Epoch 7 / 10) train acc: 0.553000; val_acc: 0.525000
(Iteration 3501 / 4900) loss: 1.346793
(Iteration 3601 / 4900) loss: 1.086192
(Iteration 3701 / 4900) loss: 1.129544
(Iteration 3801 / 4900) loss: 1.132338
(Iteration 3901 / 4900) loss: 1.154372
(Epoch 8 / 10) train acc: 0.574000; val_acc: 0.499000
(Iteration 4001 / 4900) loss: 1.081451
(Iteration 4101 / 4900) loss: 1.171715
(Iteration 4201 / 4900) loss: 1.224723
(Iteration 4301 / 4900) loss: 1.084871
(Iteration 4401 / 4900) loss: 1.009633
(Epoch 9 / 10) train acc: 0.566000; val_acc: 0.491000
(Iteration 4501 / 4900) loss: 1.098911
(Iteration 4601 / 4900) loss: 1.256289
```

```
(Iteration 4701 / 4900) loss: 1.193928
(Iteration 4801 / 4900) loss: 1.035830
(Epoch 10 / 10) train acc: 0.577000; val_acc: 0.506000
```

In [47]: *# Run this cell to visualize training loss and train / val accuracy*

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

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



Multilayer network

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

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

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

Initial loss and gradient check

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

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

```

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

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

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

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

Running check with reg = 0
Initial loss: 2.3004790897684924
W1 relative error: 7.70e-08
W2 relative error: 1.71e-05
W3 relative error: 2.95e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 6.60e-11
Running check with reg = 3.14
Initial loss: 2.3016482157750753
W1 relative error: 3.44e-07
W2 relative error: 1.48e-06
W3 relative error: 1.40e-07
b1 relative error: 2.01e-09
b2 relative error: 1.72e-09
b3 relative error: 1.32e-10

```

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

```
In [18]: # TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

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

weight_scale = 1e-2
learning_rate = 1e-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()

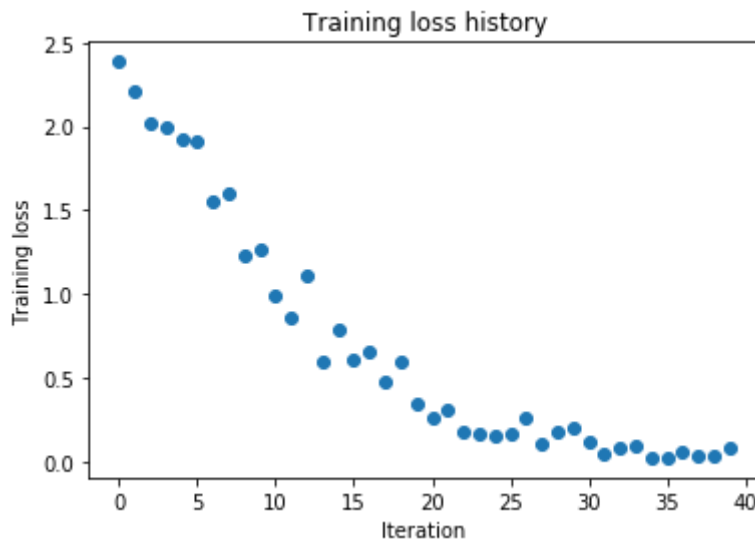
plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```



```

(Iteration 1 / 40) loss: 2.385904
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.112000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.113000
(Epoch 2 / 20) train acc: 0.480000; val_acc: 0.109000
(Epoch 3 / 20) train acc: 0.680000; val_acc: 0.157000
(Epoch 4 / 20) train acc: 0.600000; val_acc: 0.123000
(Epoch 5 / 20) train acc: 0.700000; val_acc: 0.144000
(Iteration 11 / 40) loss: 0.986071
(Epoch 6 / 20) train acc: 0.740000; val_acc: 0.153000
(Epoch 7 / 20) train acc: 0.820000; val_acc: 0.168000
(Epoch 8 / 20) train acc: 0.900000; val_acc: 0.172000
(Epoch 9 / 20) train acc: 0.920000; val_acc: 0.187000
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.160000
(Iteration 21 / 40) loss: 0.258918
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.171000
(Epoch 12 / 20) train acc: 0.920000; val_acc: 0.182000
(Epoch 13 / 20) train acc: 0.980000; val_acc: 0.189000
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.194000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.177000
(Iteration 31 / 40) loss: 0.117595
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.174000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.188000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.189000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.181000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.182000

```



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 [49]: # 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'],
}

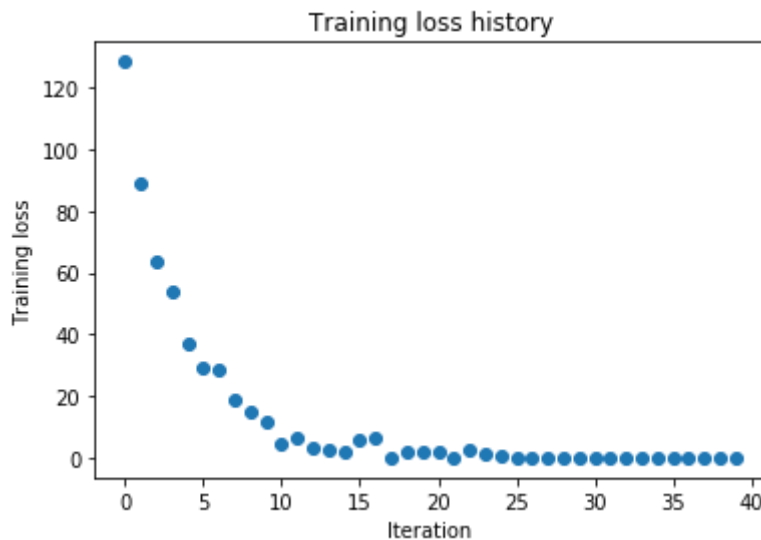
learning_rate = 3e-4 #3e-4 also works, weight_scale matters more
weight_scale = 1e-1
model = FullyConnectedNet([100, 100, 100, 100],
                           weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                 print_every=10, num_epochs=20, batch_size=25,
                 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: 128.342035
(Epoch 0 / 20) train acc: 0.200000; val_acc: 0.094000
(Epoch 1 / 20) train acc: 0.260000; val_acc: 0.091000
(Epoch 2 / 20) train acc: 0.360000; val_acc: 0.108000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.127000
(Epoch 4 / 20) train acc: 0.600000; val_acc: 0.109000
(Epoch 5 / 20) train acc: 0.640000; val_acc: 0.107000
(Iteration 11 / 40) loss: 4.724948
(Epoch 6 / 20) train acc: 0.760000; val_acc: 0.114000
(Epoch 7 / 20) train acc: 0.880000; val_acc: 0.111000
(Epoch 8 / 20) train acc: 0.900000; val_acc: 0.109000
(Epoch 9 / 20) train acc: 0.920000; val_acc: 0.107000
(Epoch 10 / 20) train acc: 0.940000; val_acc: 0.111000
(Iteration 21 / 40) loss: 1.677342
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.111000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.109000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.108000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.107000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.107000
(Iteration 31 / 40) loss: 0.001161
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.107000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.107000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.107000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.107000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.107000

```



Inline Question 2:

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

Answer:

[FILL THIS IN]

- When I was tuning three-layer vs five-layer net, I can feel the five layer seemed much more sensitive to the initialization scale, the reason behind this might be because five-layer net has more parameters which brings the loss function to a higher dimensional space compare with the loss function of three-layer net, it would be harder to find a local minimum inside a higher dimensional space rather than a less dimensional one. In this case, the start point which is settled by the initialization becomes more important for five-layer net than three-layer net.

Update rules

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

SGD+Momentum

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

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

```

In [52]: 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 [53]: num_train = 4000
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

solvers = {}

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

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

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

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

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

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

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

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

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

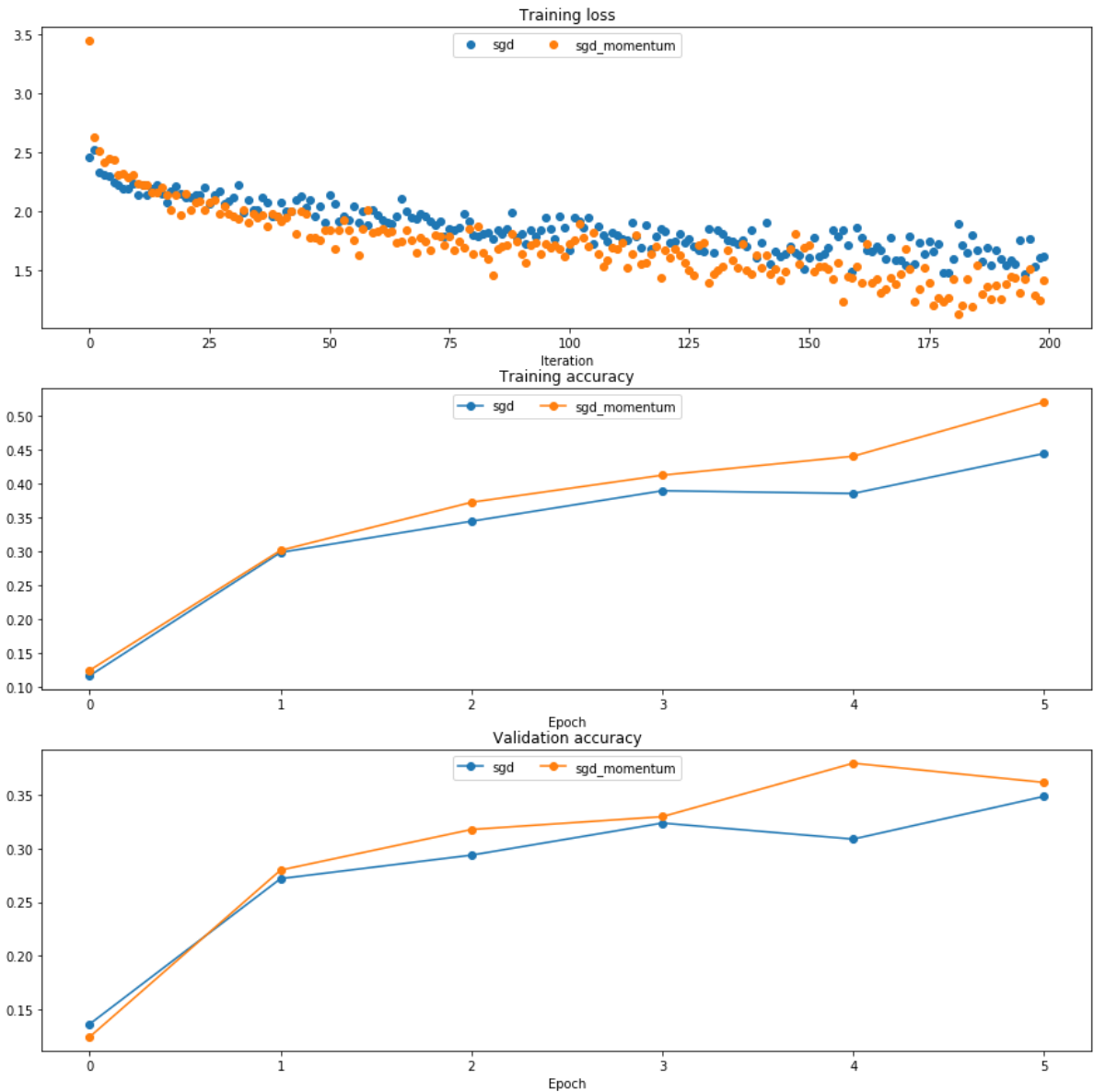
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.461307
(Epoch 0 / 5) train acc: 0.116000; val_acc: 0.136000
(Iteration 11 / 200) loss: 2.141628
(Iteration 21 / 200) loss: 2.118740
(Iteration 31 / 200) loss: 2.116650
(Epoch 1 / 5) train acc: 0.298000; val_acc: 0.272000
(Iteration 41 / 200) loss: 2.077602
(Iteration 51 / 200) loss: 2.137507
(Iteration 61 / 200) loss: 1.967009
(Iteration 71 / 200) loss: 1.957150
(Epoch 2 / 5) train acc: 0.344000; val_acc: 0.294000
(Iteration 81 / 200) loss: 1.795341
(Iteration 91 / 200) loss: 1.811299
(Iteration 101 / 200) loss: 1.666950
(Iteration 111 / 200) loss: 1.794637
(Epoch 3 / 5) train acc: 0.389000; val_acc: 0.324000
(Iteration 121 / 200) loss: 1.823884
(Iteration 131 / 200) loss: 1.652587
(Iteration 141 / 200) loss: 1.729993
(Iteration 151 / 200) loss: 1.600637
(Epoch 4 / 5) train acc: 0.385000; val_acc: 0.309000
(Iteration 161 / 200) loss: 1.859414
(Iteration 171 / 200) loss: 1.539706
(Iteration 181 / 200) loss: 1.592854
(Iteration 191 / 200) loss: 1.594035
(Epoch 5 / 5) train acc: 0.444000; val_acc: 0.349000
```

```
running with  sgd_momentum
(Iteration 1 / 200) loss: 3.446429
(Epoch 0 / 5) train acc: 0.124000; val_acc: 0.124000
(Iteration 11 / 200) loss: 2.234129
(Iteration 21 / 200) loss: 2.147259
(Iteration 31 / 200) loss: 1.952910
(Epoch 1 / 5) train acc: 0.301000; val_acc: 0.280000
(Iteration 41 / 200) loss: 1.913535
(Iteration 51 / 200) loss: 1.835153
(Iteration 61 / 200) loss: 1.824210
(Iteration 71 / 200) loss: 1.745242
(Epoch 2 / 5) train acc: 0.372000; val_acc: 0.318000
(Iteration 81 / 200) loss: 1.632745
(Iteration 91 / 200) loss: 1.636300
(Iteration 101 / 200) loss: 1.724007
(Iteration 111 / 200) loss: 1.675193
(Epoch 3 / 5) train acc: 0.412000; val_acc: 0.330000
(Iteration 121 / 200) loss: 1.673507
(Iteration 131 / 200) loss: 1.469310
(Iteration 141 / 200) loss: 1.517108
(Iteration 151 / 200) loss: 1.712278
(Epoch 4 / 5) train acc: 0.440000; val_acc: 0.380000
(Iteration 161 / 200) loss: 1.533072
(Iteration 171 / 200) loss: 1.682026
(Iteration 181 / 200) loss: 1.421417
(Iteration 191 / 200) loss: 1.256191
(Epoch 5 / 5) train acc: 0.520000; val_acc: 0.362000
```

/home/shared/anaconda3/lib/python3.6/site-packages/matplotlib/cbook/deprecation.py:106: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
warnings.warn(message, mplDeprecation, stacklevel=1)
```



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 [58]: # 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 [59]: # 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 [60]: learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
    print('running with ', update_rule)
    model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2
    )

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

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

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

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

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

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

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

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

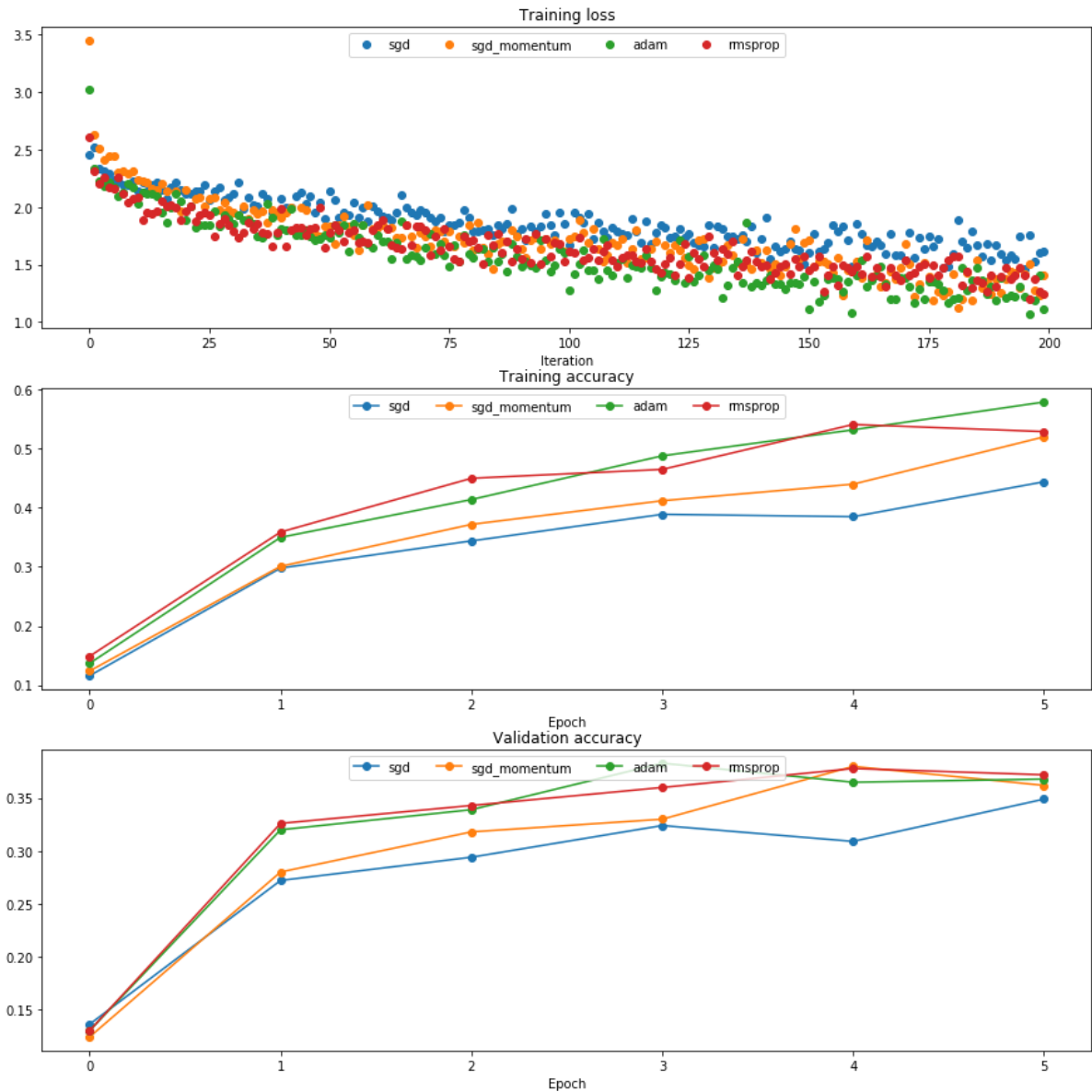
```

```
running with adam
(Iteration 1 / 200) loss: 3.020565
(Epoch 0 / 5) train acc: 0.137000; val_acc: 0.130000
(Iteration 11 / 200) loss: 2.034025
(Iteration 21 / 200) loss: 1.893583
(Iteration 31 / 200) loss: 1.853560
(Epoch 1 / 5) train acc: 0.350000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.813628
(Iteration 51 / 200) loss: 1.741199
(Iteration 61 / 200) loss: 1.762664
(Iteration 71 / 200) loss: 1.652290
(Epoch 2 / 5) train acc: 0.414000; val_acc: 0.339000
(Iteration 81 / 200) loss: 1.561304
(Iteration 91 / 200) loss: 1.485031
(Iteration 101 / 200) loss: 1.278744
(Iteration 111 / 200) loss: 1.513726
(Epoch 3 / 5) train acc: 0.488000; val_acc: 0.383000
(Iteration 121 / 200) loss: 1.436134
(Iteration 131 / 200) loss: 1.499589
(Iteration 141 / 200) loss: 1.323777
(Iteration 151 / 200) loss: 1.116501
(Epoch 4 / 5) train acc: 0.532000; val_acc: 0.365000
(Iteration 161 / 200) loss: 1.409581
(Iteration 171 / 200) loss: 1.382743
(Iteration 181 / 200) loss: 1.203627
(Iteration 191 / 200) loss: 1.388209
(Epoch 5 / 5) train acc: 0.579000; val_acc: 0.368000
```

```
running with rmsprop
(Iteration 1 / 200) loss: 2.606109
(Epoch 0 / 5) train acc: 0.149000; val_acc: 0.130000
(Iteration 11 / 200) loss: 2.086514
(Iteration 21 / 200) loss: 1.970716
(Iteration 31 / 200) loss: 1.793592
(Epoch 1 / 5) train acc: 0.359000; val_acc: 0.326000
(Iteration 41 / 200) loss: 1.991248
(Iteration 51 / 200) loss: 1.784102
(Iteration 61 / 200) loss: 1.846706
(Iteration 71 / 200) loss: 1.801463
(Epoch 2 / 5) train acc: 0.450000; val_acc: 0.343000
(Iteration 81 / 200) loss: 1.719256
(Iteration 91 / 200) loss: 1.547819
(Iteration 101 / 200) loss: 1.616752
(Iteration 111 / 200) loss: 1.488392
(Epoch 3 / 5) train acc: 0.465000; val_acc: 0.360000
(Iteration 121 / 200) loss: 1.504824
(Iteration 131 / 200) loss: 1.604127
(Iteration 141 / 200) loss: 1.485354
(Iteration 151 / 200) loss: 1.455871
(Epoch 4 / 5) train acc: 0.541000; val_acc: 0.378000
(Iteration 161 / 200) loss: 1.480721
(Iteration 171 / 200) loss: 1.538964
(Iteration 181 / 200) loss: 1.577810
(Iteration 191 / 200) loss: 1.401740
(Epoch 5 / 5) train acc: 0.529000; val_acc: 0.372000
```

/home/shared/anaconda3/lib/python3.6/site-packages/matplotlib/cbook/deprecation.py:106: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes currently reuses the earlier instance. In a future version, a new instance will always be created and returned. Meanwhile, this warning can be suppressed, and the future behavior ensured, by passing a unique label to each axes instance.

```
warnings.warn(message, mplDeprecation, stacklevel=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?

Answer:

Since every time the network use AdaGrad to update w , the `cache` is become bigger because of `cache += dw**2`, based on AdaGrad update rule, $w += - \text{learning_rate} * dw / (\text{np.sqrt(cache)} + \text{eps})$ where the absolute value of `cache` is increasing which makes the absolute value of `np.sqrt(cache)` is increasing which makes the absolute value of `learning_rate * dw / (np.sqrt(cache) + eps)` is decreasing which makes the updates become very small along per iteration.

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 [13]: 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.
#
#####
#####
# small_data = {
#     'X_train': data['X_train'],
#     'y_train': data['y_train'],
#     'X_val': data['X_val'],
#     'y_val': data['y_val'],
# }

model = FullyConnectedNet([100, 100, 100],
                           dropout=0.85,
                           normalization="batchnorm",
                           reg = 0.01,
                           weight_scale=3e-2)

solver = Solver(model, data,
                 num_epochs=20, batch_size=100,
                 update_rule='adam',
                 optim_config={
                     'learning_rate': 1e-3
                 },
                 lr_decay=0.95,
                 print_every=100000,
                 verbose=True)

solver.train()
best_model = solver.model
#####
#####
#
#
#####
#####
```

```
(Iteration 1 / 9800) loss: 3.797400
(Epoch 0 / 20) train acc: 0.103000; val_acc: 0.103000
(Epoch 1 / 20) train acc: 0.449000; val_acc: 0.432000
(Epoch 2 / 20) train acc: 0.459000; val_acc: 0.467000
(Epoch 3 / 20) train acc: 0.470000; val_acc: 0.462000
(Epoch 4 / 20) train acc: 0.458000; val_acc: 0.470000
(Epoch 5 / 20) train acc: 0.469000; val_acc: 0.484000
(Epoch 6 / 20) train acc: 0.523000; val_acc: 0.512000
(Epoch 7 / 20) train acc: 0.526000; val_acc: 0.494000
(Epoch 8 / 20) train acc: 0.518000; val_acc: 0.500000
(Epoch 9 / 20) train acc: 0.561000; val_acc: 0.518000
(Epoch 10 / 20) train acc: 0.529000; val_acc: 0.485000
(Epoch 11 / 20) train acc: 0.576000; val_acc: 0.516000
(Epoch 12 / 20) train acc: 0.531000; val_acc: 0.515000
(Epoch 13 / 20) train acc: 0.573000; val_acc: 0.512000
(Epoch 14 / 20) train acc: 0.539000; val_acc: 0.540000
(Epoch 15 / 20) train acc: 0.557000; val_acc: 0.521000
(Epoch 16 / 20) train acc: 0.563000; val_acc: 0.521000
(Epoch 17 / 20) train acc: 0.585000; val_acc: 0.540000
(Epoch 18 / 20) train acc: 0.591000; val_acc: 0.545000
(Epoch 19 / 20) train acc: 0.568000; val_acc: 0.520000
(Epoch 20 / 20) train acc: 0.576000; val_acc: 0.543000
```

Test your model!

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

```
In [14]: 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: 0.533
Test set accuracy: 0.505
```


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 [55]: # 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_array
from cs231n.solver import Solver

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

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

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

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 [56]: # 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,)
```

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 [85]: # 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:  [4.66293670e-17 1.66533454e-17 1.85962357e-17]
stds:   [0.99999999 1.          1.          ]

```

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.])

```

means:  [11. 12. 13.]
stds:   [0.99999999 1.99999999 2.99999999]

```

```
In [86]: # 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]
```

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 [112]: # 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:  1.7029258328157158e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

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 [115]: 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:  9.20004371222927e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.34x
```

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

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
# and losses between 1e-08~1e-09 for beta and gammas.
for reg in [0, 3.14]:
    print('Running check with reg = ', reg)
    model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                              reg=reg, weight_scale=5e-2, dtype=np.float64,
                              normalization='batchnorm')

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

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

```
Running check with reg = 0
Initial loss: 2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 2.85e-06
W3 relative error: 4.05e-10
b1 relative error: 4.44e-08
b2 relative error: 2.22e-08
b3 relative error: 1.01e-10
beta1 relative error: 7.33e-09
beta2 relative error: 1.89e-09
gamma1 relative error: 6.96e-09
gamma2 relative error: 1.96e-09
```

```
Running check with reg = 3.14
Initial loss: 6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.29e-06
W3 relative error: 2.79e-08
b1 relative error: 2.78e-09
b2 relative error: 2.22e-08
b3 relative error: 2.10e-10
beta1 relative error: 6.65e-09
beta2 relative error: 4.23e-09
gamma1 relative error: 6.27e-09
gamma2 relative error: 5.28e-09
```


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

num_train = 1000
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

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

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

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

```
(Iteration 1 / 200) loss: 2.340975
(Epoch 0 / 10) train acc: 0.101000; val_acc: 0.112000
(Epoch 1 / 10) train acc: 0.246000; val_acc: 0.215000
(Iteration 21 / 200) loss: 1.914006
(Epoch 2 / 10) train acc: 0.337000; val_acc: 0.290000
(Iteration 41 / 200) loss: 2.119071
(Epoch 3 / 10) train acc: 0.412000; val_acc: 0.295000
(Iteration 61 / 200) loss: 1.927924
(Epoch 4 / 10) train acc: 0.440000; val_acc: 0.313000
(Iteration 81 / 200) loss: 1.526590
(Epoch 5 / 10) train acc: 0.473000; val_acc: 0.296000
(Iteration 101 / 200) loss: 1.412473
(Epoch 6 / 10) train acc: 0.541000; val_acc: 0.320000
(Iteration 121 / 200) loss: 1.341697
(Epoch 7 / 10) train acc: 0.598000; val_acc: 0.342000
(Iteration 141 / 200) loss: 1.446070
(Epoch 8 / 10) train acc: 0.646000; val_acc: 0.341000
(Iteration 161 / 200) loss: 0.843562
(Epoch 9 / 10) train acc: 0.715000; val_acc: 0.342000
(Iteration 181 / 200) loss: 1.050745
(Epoch 10 / 10) train acc: 0.727000; val_acc: 0.337000
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.155000; val_acc: 0.140000
(Epoch 1 / 10) train acc: 0.182000; val_acc: 0.160000
(Iteration 21 / 200) loss: 2.143171
(Epoch 2 / 10) train acc: 0.212000; val_acc: 0.178000
(Iteration 41 / 200) loss: 1.915787
(Epoch 3 / 10) train acc: 0.244000; val_acc: 0.228000
(Iteration 61 / 200) loss: 1.856857
(Epoch 4 / 10) train acc: 0.279000; val_acc: 0.255000
(Iteration 81 / 200) loss: 1.789520
(Epoch 5 / 10) train acc: 0.290000; val_acc: 0.253000
(Iteration 101 / 200) loss: 1.839935
(Epoch 6 / 10) train acc: 0.337000; val_acc: 0.252000
(Iteration 121 / 200) loss: 1.793666
(Epoch 7 / 10) train acc: 0.370000; val_acc: 0.252000
(Iteration 141 / 200) loss: 1.725922
(Epoch 8 / 10) train acc: 0.359000; val_acc: 0.254000
(Iteration 161 / 200) loss: 1.691867
(Epoch 9 / 10) train acc: 0.405000; val_acc: 0.261000
(Iteration 181 / 200) loss: 1.576471
(Epoch 10 / 10) train acc: 0.415000; val_acc: 0.277000
```

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.

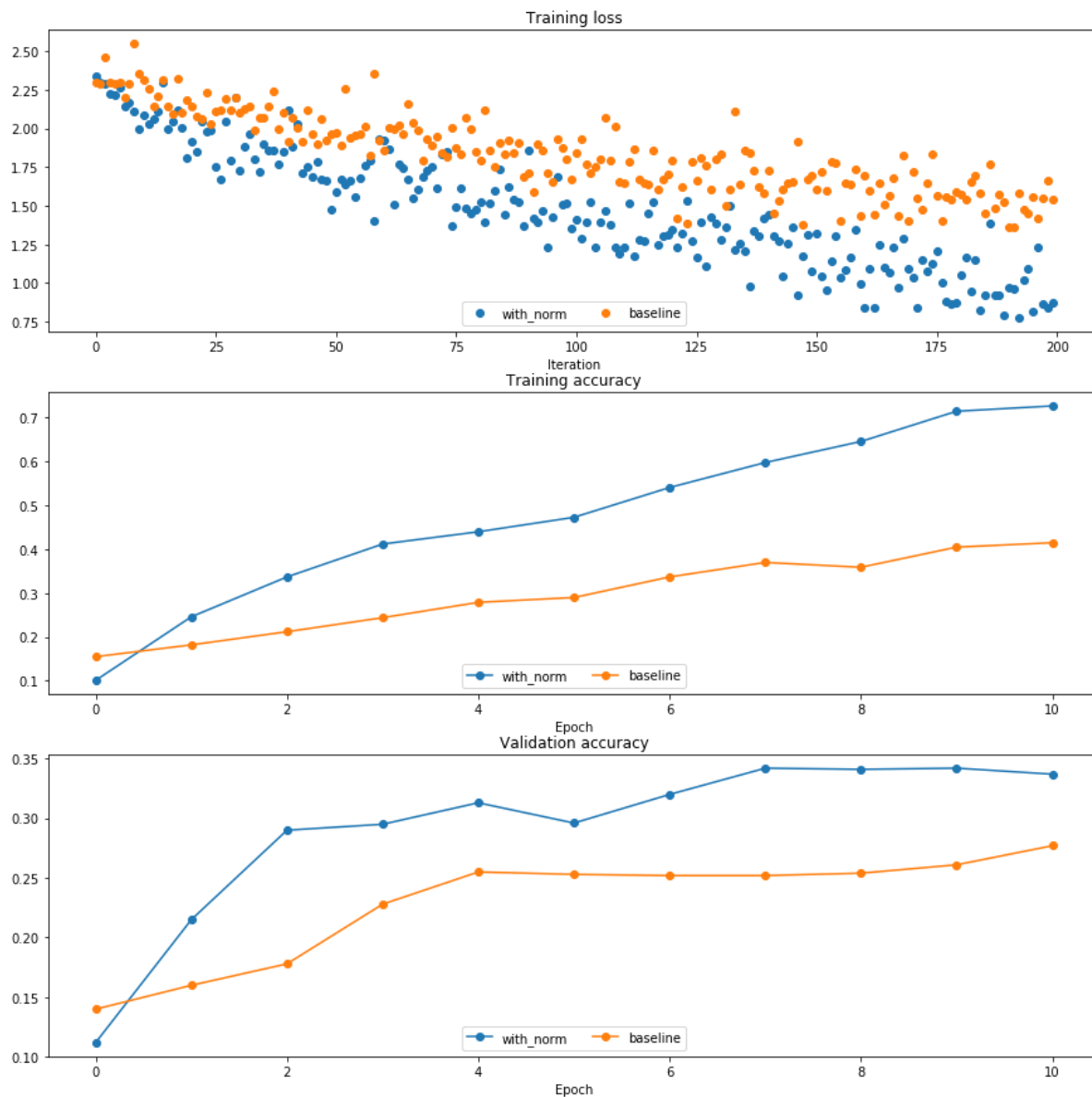
```

In [126]: def plot_training_history(title, label, baseline, bn_solvers, plot_fn, b
l_marker='.', bn_marker='.', labels=None):
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)

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

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

```



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 [127]: np.random.seed(231)
# Try training a very deep net with batchnorm
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
small_data = {
    'X_train': data['X_train'][:num_train],
    'y_train': data['y_train'][:num_train],
    'X_val': data['X_val'],
    'y_val': data['y_val'],
}

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

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

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

```

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

```
In [128]: # 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))

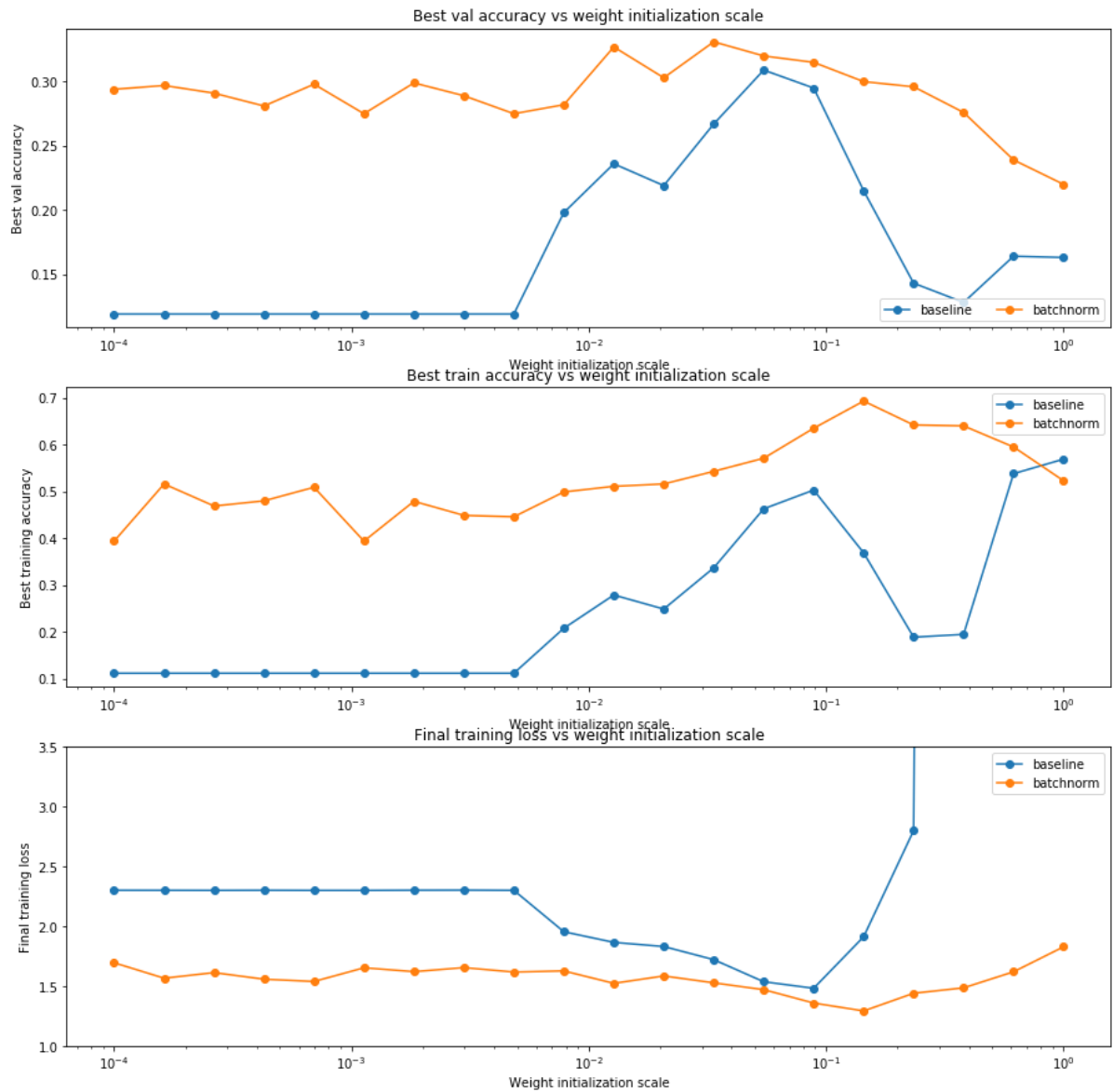
    final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
    bn_final_train_loss.append(np.mean(bn_solvers_ws[ws].loss_history[-100:]))

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()
```

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?

Answer:

As we can see, the performance are very sensitive to the scale of weight initialization when the model train/evaluate without batch normalization. It became much stable and works well when the model train/evaluate with batch normalization, tuning scale of weight initialization seems not affect its performance at all after put batch normalization. By comparing the best performance we could also see models with batch normalization works better than the one without batch normalization

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 [129]: 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, no
rmalization=None)
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='adam',
                    optim_config={
                        'learning_rate': lr,
                    },
                    verbose=False)

    solver.train()

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

        bn_solver.train()
        bn_solvers.append(bn_solver)

    return bn_solvers, solver, batch_sizes

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

```

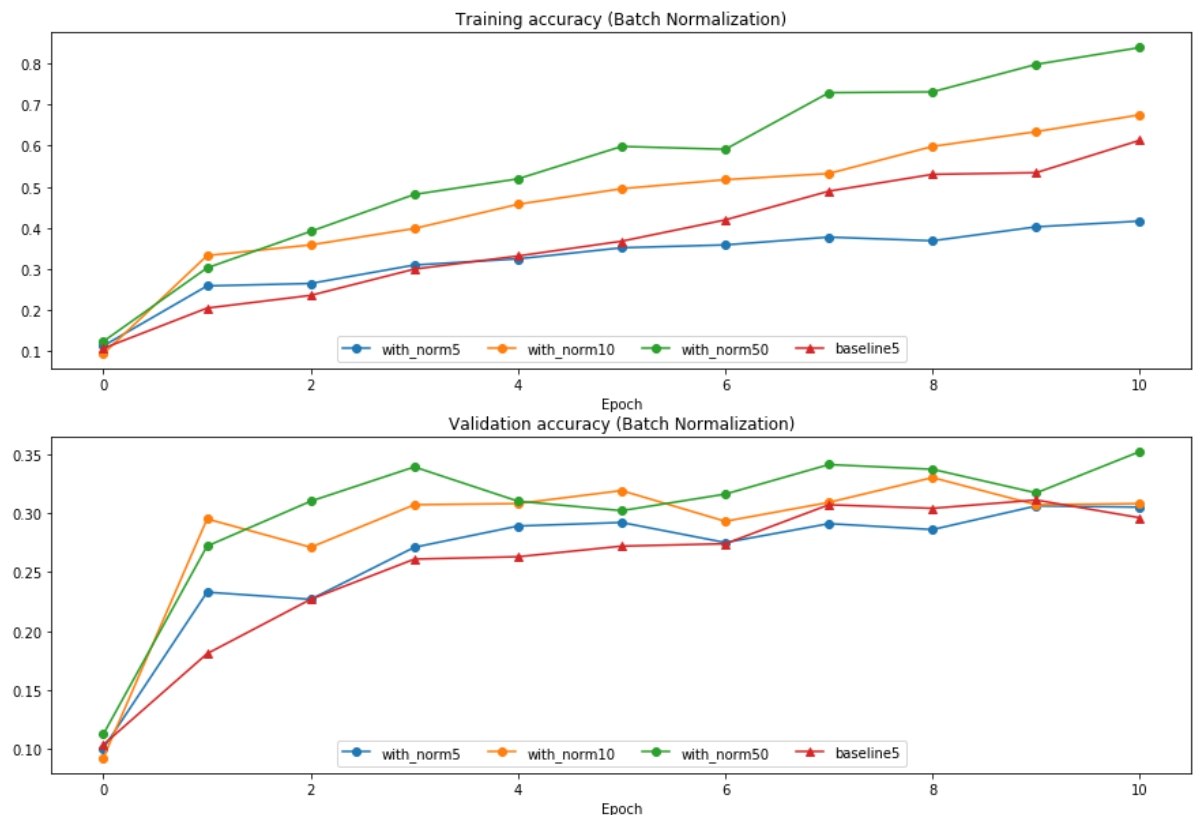
```

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

```

```
In [130]: plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)', 'Epoch',
solver_bsize, bn_solvers_bsize, \
                    lambda x: x.train_acc_history, bl_marker='^-', bn_
marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)', 'Epoch',
solver_bsize, bn_solvers_bsize, \
                    lambda x: x.val_acc_history, bl_marker='^-', bn_ma
rker='-o', labels=batch_sizes)

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



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?

Answer:

The larger batch size we use for batch normalization, the better accuracy we could get for training and validation. the reason behind this would be when we use batch normalization, we normalize the input X by the mean and variance of corresponding mini batch, the larger the mini batch are, the better we could normalize the input X which would help to increase the train/validation accuracy

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>)

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.

Answer:

1. layer
2. layer
3. batch
4. batch

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

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

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

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

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

Before layer normalization:

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

After layer normalization (gamma=1, beta=0)

```
means: [ 4.81096644e-16 -7.40148683e-17  2.22044605e-16 -5.92118946e-16]
stds:  [0.99999995 0.99999999 1.          0.99999969]
```

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.])

```
means: [5. 5. 5. 5.]
stds:  [2.99999985 2.99999998 2.99999999 2.99999907]
```

```

In [141]: # 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:  1.433615657860454e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12

```

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 [142]: ln_solvers_bsize, solver_bsize, batch_sizes = run_batchsize_experiments(
'layernorm')

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

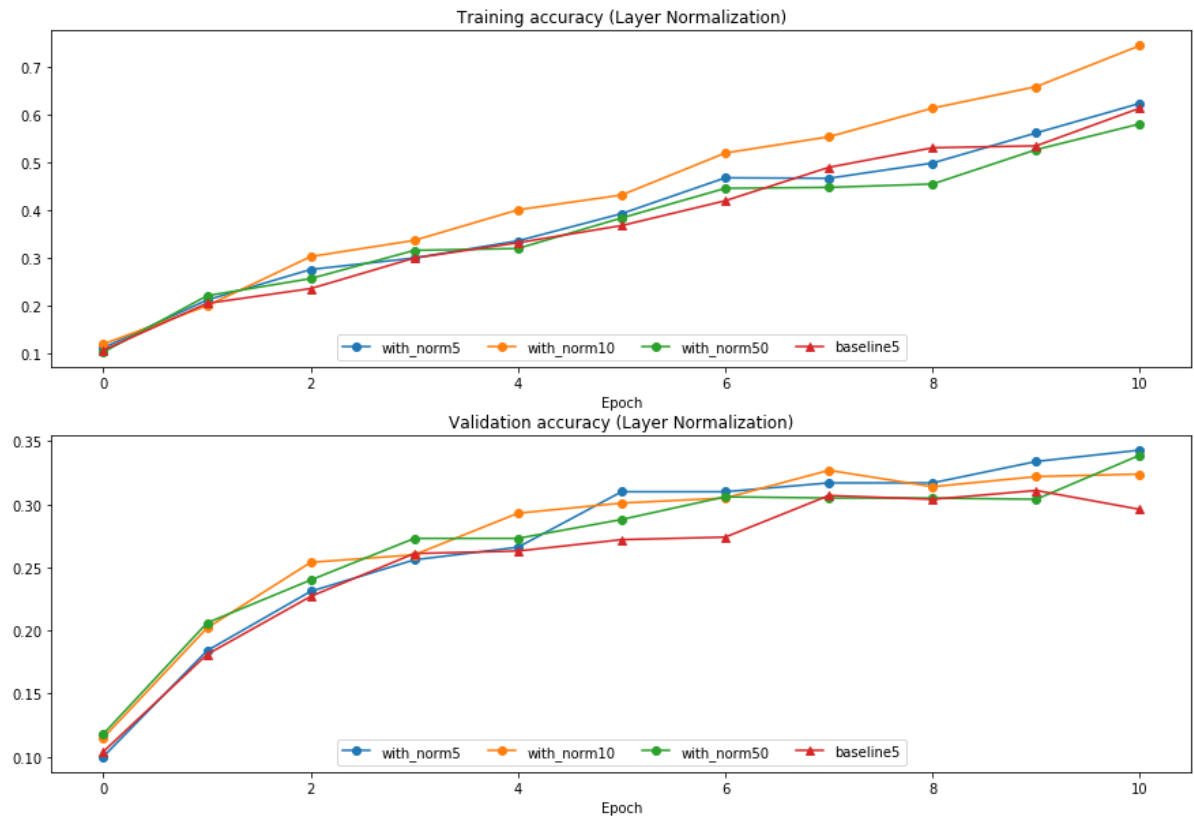
plt.gcf().set_size_inches(15, 10)
plt.show()
```

No normalization: batch size = 5

Normalization: batch size = 5

Normalization: batch size = 10

Normalization: batch size = 50



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

Answer:

2. Having a very small dimension of features

- Layer normalization having a very small dimension of features is like batch normalization having a very small size of batch, both of which won't normalize the input X very well because the mean and variance is calculated based on a very small size of batch/features

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](https://arxiv.org/abs/1207.0580)", *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_array
from cs231n.solver import Solver

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

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

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

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,)
```

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 [24]: 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).me
    an())
    print()
```

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

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

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

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

dx relative error: 1.892896957390533e-11
```

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?

Answer:

The model will show overfitting during testing. Because during training, if we do not divide the values by p , to still use dropout, we will have to scale the activations by p during testing, if we don't scale the activations by p during testing, it means we are using all neurons (expect neurons $\times p$) which will definitely cause overfitting.

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

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

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

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

```

```

Running check with dropout = 1
Initial loss: 2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11

```

```

Running check with dropout = 0.75
Initial loss: 2.3016482157750753
W1 relative error: 6.96e-07
W2 relative error: 5.01e-06
W3 relative error: 2.96e-07
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.32e-10

```

```

Running check with dropout = 0.5
Initial loss: 2.30485979047391
W1 relative error: 8.77e-07
W2 relative error: 3.64e-07
W3 relative error: 6.33e-08
b1 relative error: 1.22e-07
b2 relative error: 4.50e-09
b3 relative error: 1.15e-10

```

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 [39]: # 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, 0.95, 0.75, 0.5, 0.25, 0.1, 0.01]
for dropout in dropout_choices:
    model = FullyConnectedNet([500], dropout=dropout)
    print(dropout)

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

1

```
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.236000; val_acc: 0.190000
(Epoch 1 / 25) train acc: 0.250000; val_acc: 0.178000
(Epoch 2 / 25) train acc: 0.360000; val_acc: 0.217000
(Epoch 3 / 25) train acc: 0.508000; val_acc: 0.242000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.244000
(Epoch 5 / 25) train acc: 0.576000; val_acc: 0.288000
(Epoch 6 / 25) train acc: 0.596000; val_acc: 0.263000
(Epoch 7 / 25) train acc: 0.698000; val_acc: 0.245000
(Epoch 8 / 25) train acc: 0.766000; val_acc: 0.285000
(Epoch 9 / 25) train acc: 0.822000; val_acc: 0.304000
(Epoch 10 / 25) train acc: 0.844000; val_acc: 0.294000
(Epoch 11 / 25) train acc: 0.888000; val_acc: 0.301000
(Epoch 12 / 25) train acc: 0.914000; val_acc: 0.291000
(Epoch 13 / 25) train acc: 0.926000; val_acc: 0.309000
(Epoch 14 / 25) train acc: 0.938000; val_acc: 0.314000
(Epoch 15 / 25) train acc: 0.942000; val_acc: 0.301000
(Epoch 16 / 25) train acc: 0.942000; val_acc: 0.291000
(Epoch 17 / 25) train acc: 0.926000; val_acc: 0.289000
(Epoch 18 / 25) train acc: 0.938000; val_acc: 0.305000
(Epoch 19 / 25) train acc: 0.972000; val_acc: 0.301000
(Epoch 20 / 25) train acc: 0.986000; val_acc: 0.283000
(Iteration 101 / 125) loss: 0.000024
(Epoch 21 / 25) train acc: 0.980000; val_acc: 0.308000
(Epoch 22 / 25) train acc: 0.978000; val_acc: 0.306000
(Epoch 23 / 25) train acc: 0.986000; val_acc: 0.309000
(Epoch 24 / 25) train acc: 0.974000; val_acc: 0.302000
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.302000
0.95
```

```
(Iteration 1 / 125) loss: 10.430469
(Epoch 0 / 25) train acc: 0.238000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.268000; val_acc: 0.186000
(Epoch 2 / 25) train acc: 0.356000; val_acc: 0.225000
(Epoch 3 / 25) train acc: 0.378000; val_acc: 0.189000
(Epoch 4 / 25) train acc: 0.458000; val_acc: 0.226000
(Epoch 5 / 25) train acc: 0.570000; val_acc: 0.271000
(Epoch 6 / 25) train acc: 0.598000; val_acc: 0.260000
(Epoch 7 / 25) train acc: 0.714000; val_acc: 0.262000
(Epoch 8 / 25) train acc: 0.760000; val_acc: 0.269000
(Epoch 9 / 25) train acc: 0.828000; val_acc: 0.285000
(Epoch 10 / 25) train acc: 0.832000; val_acc: 0.289000
(Epoch 11 / 25) train acc: 0.856000; val_acc: 0.299000
(Epoch 12 / 25) train acc: 0.846000; val_acc: 0.290000
(Epoch 13 / 25) train acc: 0.880000; val_acc: 0.288000
(Epoch 14 / 25) train acc: 0.898000; val_acc: 0.273000
(Epoch 15 / 25) train acc: 0.936000; val_acc: 0.289000
(Epoch 16 / 25) train acc: 0.946000; val_acc: 0.289000
(Epoch 17 / 25) train acc: 0.942000; val_acc: 0.292000
(Epoch 18 / 25) train acc: 0.968000; val_acc: 0.298000
(Epoch 19 / 25) train acc: 0.974000; val_acc: 0.305000
(Epoch 20 / 25) train acc: 0.972000; val_acc: 0.295000
(Iteration 101 / 125) loss: 0.149532
(Epoch 21 / 25) train acc: 0.972000; val_acc: 0.305000
(Epoch 22 / 25) train acc: 0.964000; val_acc: 0.307000
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.281000
(Epoch 24 / 25) train acc: 0.978000; val_acc: 0.297000
```

```
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.295000
0.75
(Iteration 1 / 125) loss: 10.766073
(Epoch 0 / 25) train acc: 0.228000; val_acc: 0.179000
(Epoch 1 / 25) train acc: 0.278000; val_acc: 0.191000
(Epoch 2 / 25) train acc: 0.346000; val_acc: 0.233000
(Epoch 3 / 25) train acc: 0.384000; val_acc: 0.225000
(Epoch 4 / 25) train acc: 0.440000; val_acc: 0.240000
(Epoch 5 / 25) train acc: 0.528000; val_acc: 0.272000
(Epoch 6 / 25) train acc: 0.630000; val_acc: 0.259000
(Epoch 7 / 25) train acc: 0.688000; val_acc: 0.284000
(Epoch 8 / 25) train acc: 0.740000; val_acc: 0.283000
(Epoch 9 / 25) train acc: 0.830000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.808000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.866000; val_acc: 0.281000
(Epoch 12 / 25) train acc: 0.900000; val_acc: 0.285000
(Epoch 13 / 25) train acc: 0.910000; val_acc: 0.297000
(Epoch 14 / 25) train acc: 0.914000; val_acc: 0.290000
(Epoch 15 / 25) train acc: 0.910000; val_acc: 0.268000
(Epoch 16 / 25) train acc: 0.934000; val_acc: 0.288000
(Epoch 17 / 25) train acc: 0.932000; val_acc: 0.269000
(Epoch 18 / 25) train acc: 0.930000; val_acc: 0.305000
(Epoch 19 / 25) train acc: 0.944000; val_acc: 0.288000
(Epoch 20 / 25) train acc: 0.968000; val_acc: 0.292000
(Iteration 101 / 125) loss: 0.155204
(Epoch 21 / 25) train acc: 0.972000; val_acc: 0.280000
(Epoch 22 / 25) train acc: 0.988000; val_acc: 0.294000
(Epoch 23 / 25) train acc: 0.990000; val_acc: 0.298000
(Epoch 24 / 25) train acc: 0.988000; val_acc: 0.296000
(Epoch 25 / 25) train acc: 0.994000; val_acc: 0.304000
0.5
(Iteration 1 / 125) loss: 10.290508
(Epoch 0 / 25) train acc: 0.254000; val_acc: 0.144000
(Epoch 1 / 25) train acc: 0.220000; val_acc: 0.165000
(Epoch 2 / 25) train acc: 0.314000; val_acc: 0.229000
(Epoch 3 / 25) train acc: 0.456000; val_acc: 0.274000
(Epoch 4 / 25) train acc: 0.552000; val_acc: 0.302000
(Epoch 5 / 25) train acc: 0.608000; val_acc: 0.307000
(Epoch 6 / 25) train acc: 0.624000; val_acc: 0.276000
(Epoch 7 / 25) train acc: 0.690000; val_acc: 0.314000
(Epoch 8 / 25) train acc: 0.704000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.782000; val_acc: 0.310000
(Epoch 10 / 25) train acc: 0.820000; val_acc: 0.302000
(Epoch 11 / 25) train acc: 0.872000; val_acc: 0.274000
(Epoch 12 / 25) train acc: 0.918000; val_acc: 0.281000
(Epoch 13 / 25) train acc: 0.924000; val_acc: 0.303000
(Epoch 14 / 25) train acc: 0.942000; val_acc: 0.299000
(Epoch 15 / 25) train acc: 0.918000; val_acc: 0.289000
(Epoch 16 / 25) train acc: 0.948000; val_acc: 0.301000
(Epoch 17 / 25) train acc: 0.962000; val_acc: 0.287000
(Epoch 18 / 25) train acc: 0.956000; val_acc: 0.293000
(Epoch 19 / 25) train acc: 0.950000; val_acc: 0.283000
(Epoch 20 / 25) train acc: 0.980000; val_acc: 0.284000
(Iteration 101 / 125) loss: 0.599464
(Epoch 21 / 25) train acc: 0.968000; val_acc: 0.297000
(Epoch 22 / 25) train acc: 0.970000; val_acc: 0.297000
(Epoch 23 / 25) train acc: 0.982000; val_acc: 0.304000
```

```
(Epoch 24 / 25) train acc: 0.994000; val_acc: 0.304000
(Epoch 25 / 25) train acc: 0.992000; val_acc: 0.306000
0.25
(Iteration 1 / 125) loss: 8.286989
(Epoch 0 / 25) train acc: 0.192000; val_acc: 0.160000
(Epoch 1 / 25) train acc: 0.272000; val_acc: 0.201000
(Epoch 2 / 25) train acc: 0.360000; val_acc: 0.203000
(Epoch 3 / 25) train acc: 0.486000; val_acc: 0.249000
(Epoch 4 / 25) train acc: 0.540000; val_acc: 0.263000
(Epoch 5 / 25) train acc: 0.558000; val_acc: 0.260000
(Epoch 6 / 25) train acc: 0.606000; val_acc: 0.243000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.233000
(Epoch 8 / 25) train acc: 0.712000; val_acc: 0.272000
(Epoch 9 / 25) train acc: 0.770000; val_acc: 0.270000
(Epoch 10 / 25) train acc: 0.784000; val_acc: 0.288000
(Epoch 11 / 25) train acc: 0.856000; val_acc: 0.263000
(Epoch 12 / 25) train acc: 0.922000; val_acc: 0.295000
(Epoch 13 / 25) train acc: 0.892000; val_acc: 0.278000
(Epoch 14 / 25) train acc: 0.878000; val_acc: 0.293000
(Epoch 15 / 25) train acc: 0.926000; val_acc: 0.284000
(Epoch 16 / 25) train acc: 0.934000; val_acc: 0.304000
(Epoch 17 / 25) train acc: 0.940000; val_acc: 0.301000
(Epoch 18 / 25) train acc: 0.958000; val_acc: 0.301000
(Epoch 19 / 25) train acc: 0.982000; val_acc: 0.293000
(Epoch 20 / 25) train acc: 0.976000; val_acc: 0.287000
(Iteration 101 / 125) loss: 1.150880
(Epoch 21 / 25) train acc: 0.970000; val_acc: 0.300000
(Epoch 22 / 25) train acc: 0.992000; val_acc: 0.299000
(Epoch 23 / 25) train acc: 0.974000; val_acc: 0.298000
(Epoch 24 / 25) train acc: 0.988000; val_acc: 0.311000
(Epoch 25 / 25) train acc: 0.974000; val_acc: 0.314000
0.1
(Iteration 1 / 125) loss: 7.980158
(Epoch 0 / 25) train acc: 0.226000; val_acc: 0.176000
(Epoch 1 / 25) train acc: 0.226000; val_acc: 0.155000
(Epoch 2 / 25) train acc: 0.368000; val_acc: 0.224000
(Epoch 3 / 25) train acc: 0.458000; val_acc: 0.246000
(Epoch 4 / 25) train acc: 0.496000; val_acc: 0.256000
(Epoch 5 / 25) train acc: 0.554000; val_acc: 0.242000
(Epoch 6 / 25) train acc: 0.666000; val_acc: 0.274000
(Epoch 7 / 25) train acc: 0.696000; val_acc: 0.289000
(Epoch 8 / 25) train acc: 0.712000; val_acc: 0.270000
(Epoch 9 / 25) train acc: 0.752000; val_acc: 0.265000
(Epoch 10 / 25) train acc: 0.840000; val_acc: 0.275000
(Epoch 11 / 25) train acc: 0.852000; val_acc: 0.253000
(Epoch 12 / 25) train acc: 0.914000; val_acc: 0.272000
(Epoch 13 / 25) train acc: 0.932000; val_acc: 0.297000
(Epoch 14 / 25) train acc: 0.920000; val_acc: 0.297000
(Epoch 15 / 25) train acc: 0.956000; val_acc: 0.309000
(Epoch 16 / 25) train acc: 0.910000; val_acc: 0.306000
(Epoch 17 / 25) train acc: 0.964000; val_acc: 0.303000
(Epoch 18 / 25) train acc: 0.956000; val_acc: 0.296000
(Epoch 19 / 25) train acc: 0.966000; val_acc: 0.296000
(Epoch 20 / 25) train acc: 0.982000; val_acc: 0.293000
(Iteration 101 / 125) loss: 0.211149
(Epoch 21 / 25) train acc: 0.986000; val_acc: 0.304000
(Epoch 22 / 25) train acc: 0.980000; val_acc: 0.304000
```

```
(Epoch 23 / 25) train acc: 0.976000; val_acc: 0.303000
(Epoch 24 / 25) train acc: 0.996000; val_acc: 0.305000
(Epoch 25 / 25) train acc: 0.998000; val_acc: 0.307000
0.01
(Iteration 1 / 125) loss: 9.388144
(Epoch 0 / 25) train acc: 0.246000; val_acc: 0.211000
(Epoch 1 / 25) train acc: 0.234000; val_acc: 0.163000
(Epoch 2 / 25) train acc: 0.386000; val_acc: 0.224000
(Epoch 3 / 25) train acc: 0.456000; val_acc: 0.290000
(Epoch 4 / 25) train acc: 0.488000; val_acc: 0.257000
(Epoch 5 / 25) train acc: 0.530000; val_acc: 0.257000
(Epoch 6 / 25) train acc: 0.608000; val_acc: 0.263000
(Epoch 7 / 25) train acc: 0.734000; val_acc: 0.272000
(Epoch 8 / 25) train acc: 0.720000; val_acc: 0.282000
(Epoch 9 / 25) train acc: 0.782000; val_acc: 0.260000
(Epoch 10 / 25) train acc: 0.798000; val_acc: 0.266000
(Epoch 11 / 25) train acc: 0.826000; val_acc: 0.234000
(Epoch 12 / 25) train acc: 0.900000; val_acc: 0.293000
(Epoch 13 / 25) train acc: 0.872000; val_acc: 0.285000
(Epoch 14 / 25) train acc: 0.910000; val_acc: 0.290000
(Epoch 15 / 25) train acc: 0.896000; val_acc: 0.302000
(Epoch 16 / 25) train acc: 0.940000; val_acc: 0.285000
(Epoch 17 / 25) train acc: 0.940000; val_acc: 0.313000
(Epoch 18 / 25) train acc: 0.958000; val_acc: 0.301000
(Epoch 19 / 25) train acc: 0.968000; val_acc: 0.282000
(Epoch 20 / 25) train acc: 0.994000; val_acc: 0.292000
(Iteration 101 / 125) loss: 0.168024
(Epoch 21 / 25) train acc: 0.986000; val_acc: 0.305000
(Epoch 22 / 25) train acc: 0.984000; val_acc: 0.305000
(Epoch 23 / 25) train acc: 0.994000; val_acc: 0.291000
(Epoch 24 / 25) train acc: 0.994000; val_acc: 0.280000
(Epoch 25 / 25) train acc: 0.992000; val_acc: 0.281000
```

```

In [40]: # Plot train and validation accuracies of the two models

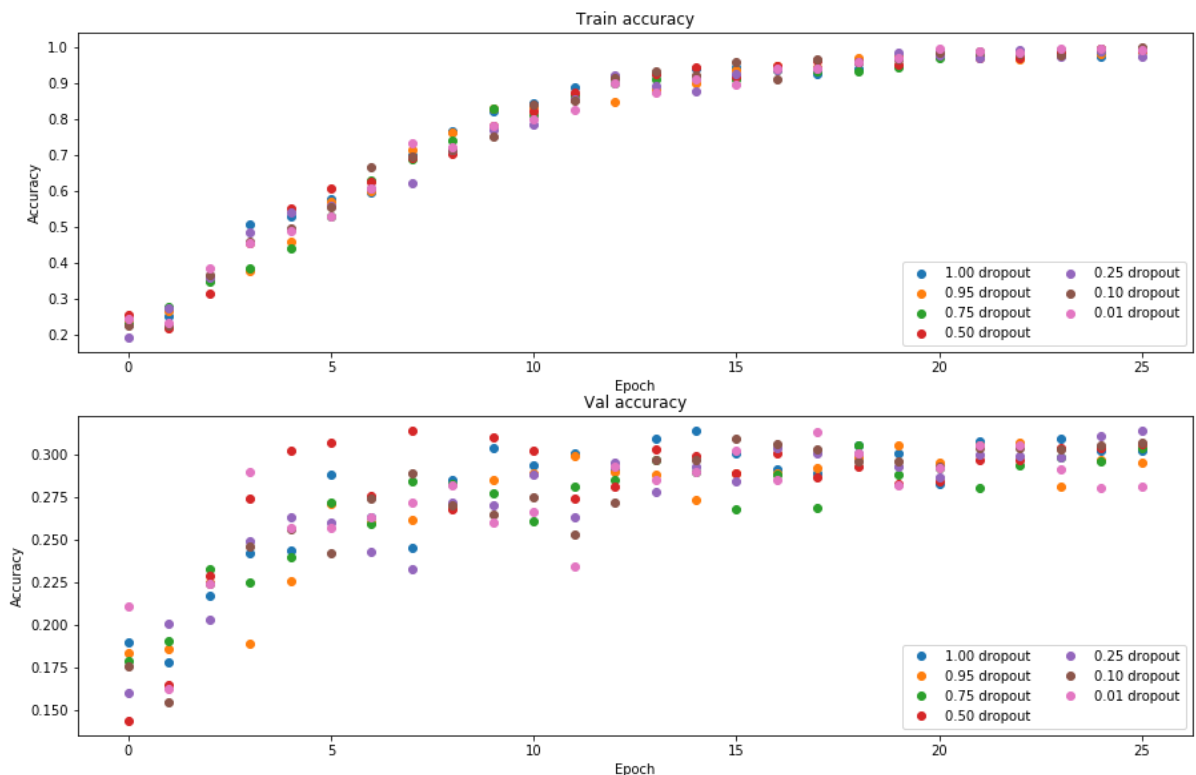
train_accs = []
val_accs = []
for dropout in dropout_choices:
    solver = solvers[dropout]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

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()

```



Inline Question 2:

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

Answer:

Dropout could prevent model from overfitting which reduce the gap between training accuracy and val accuracy, however if we use extremely high dropout rate, it would make the model less fit(see 0.01 dropout upon).

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)?

Answer:

If we decide to decrease the size of the hidden layer, we should decrease keep probability p because decrease the size of hidden layers is equal to decrease the keep probability p

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 [17]: # 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_gradient
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))))
```

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

```
%reload_ext autoreload
```



```
In [3]: # 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,)
```

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 [79]: 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
```

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 [80]: from scipy.misc import imread, imresize

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

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

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

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

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

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

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

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

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

```

```
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
/home/shared/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.p
y:3: DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

This is separate from the ipykernel package so we can avoid doing imp
orts until

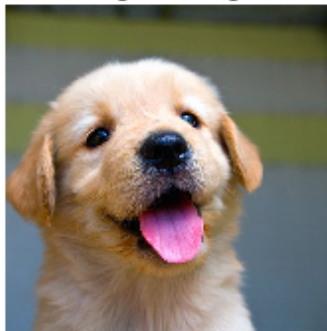
```
/home/shared/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.p
y:10: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
```

Remove the CWD from sys.path while we load stuff.

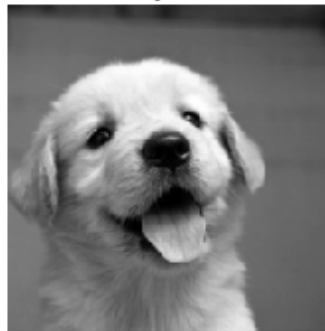
```
/home/shared/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.p
y:11: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
```

This is added back by InteractiveShellApp.init_path()

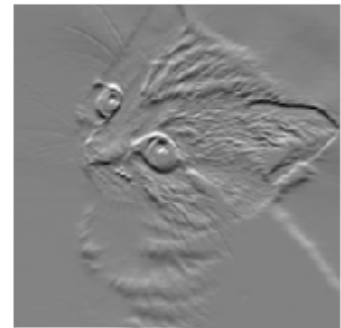
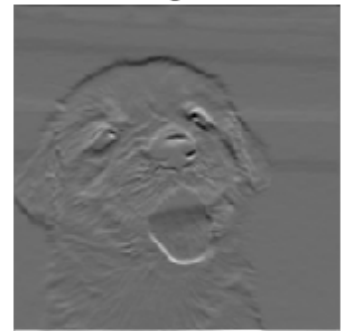
Original image



Grayscale



Edges



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 [37]: 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)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w
, b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w
, b, conv_param)[0], b, dout)

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:  2.2441985546607075e-09
dw error:  9.747020450039863e-10
db error:  4.3909013393193666e-11
```

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 [43]: 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
```

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

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

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

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

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

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 receives 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 [101]: # 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: 14.163476s
Fast: 0.009443s
Speedup: 1499.884515x
Difference: 2.955844710838216e-11

Testing conv_backward_fast:
Naive: 23.339670s
Fast: 0.007937s
Speedup: 2940.456296x
dx difference: 1.949764775345631e-11
dw difference: 3.681156828004736e-13
db difference: 0.0
```



```
In [114]: # 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: 0.374028s
fast: 0.002635s
speedup: 141.971765x
difference: 0.0
```

```
Testing pool_backward_fast:
Naive: 0.338437s
fast: 0.012281s
speedup: 27.557357x
dx difference: 0.0
```

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

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error: 9.591132621921372e-09
dw error: 5.802391137330214e-09
db error: 1.0146343411762047e-09
```

```
In [116]: 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)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w,
    b, conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w,
    b, conv_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))

Testing conv_relu:
dx error:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

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:

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 [117]: 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): 2.302586071243987
Initial loss (with regularization): 2.508255638232932
```

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 artificial 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 [118]: num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                           input_dim=input_dim, hidden_dim=7,
                           dtype=np.float64)

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

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

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 [4]: 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)

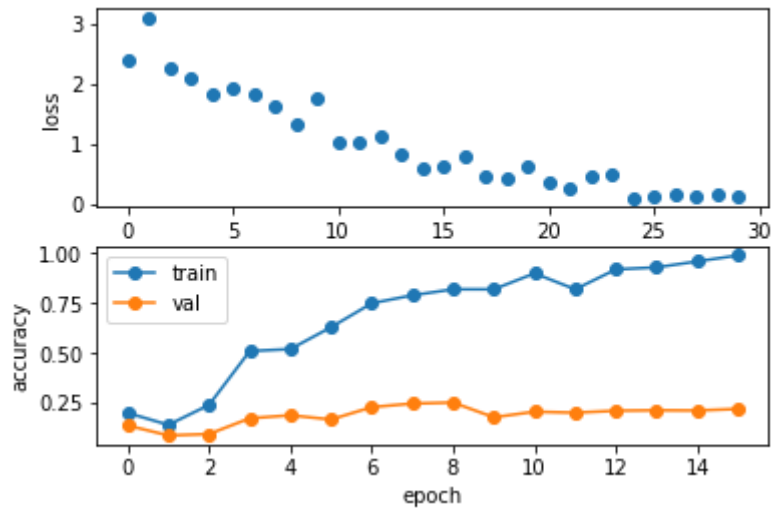
solver = Solver(model, small_data,
                 num_epochs=15, batch_size=50,
                 update_rule='adam',
                 optim_config={
                     'learning_rate': 1e-3,
                 },
                 verbose=True, print_every=1)
solver.train()
```

```
(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 [5]: 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()
```



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 [6]: 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()
```

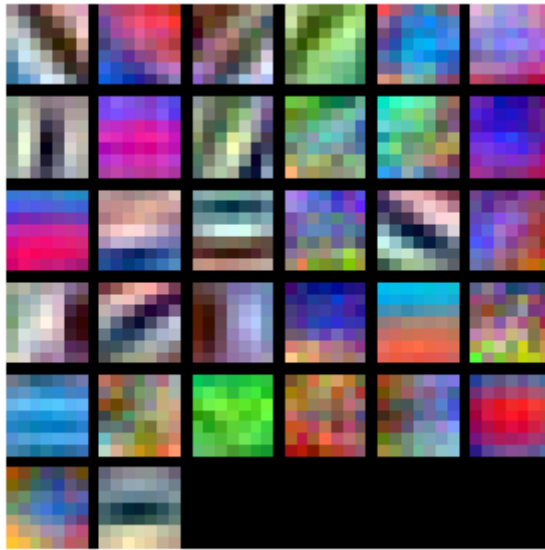
```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 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
```

Visualize Filters

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

```
In [10]: 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()
```



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 images and 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>).

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 [11]: 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 [12]: 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]

```

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 [13]: 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))

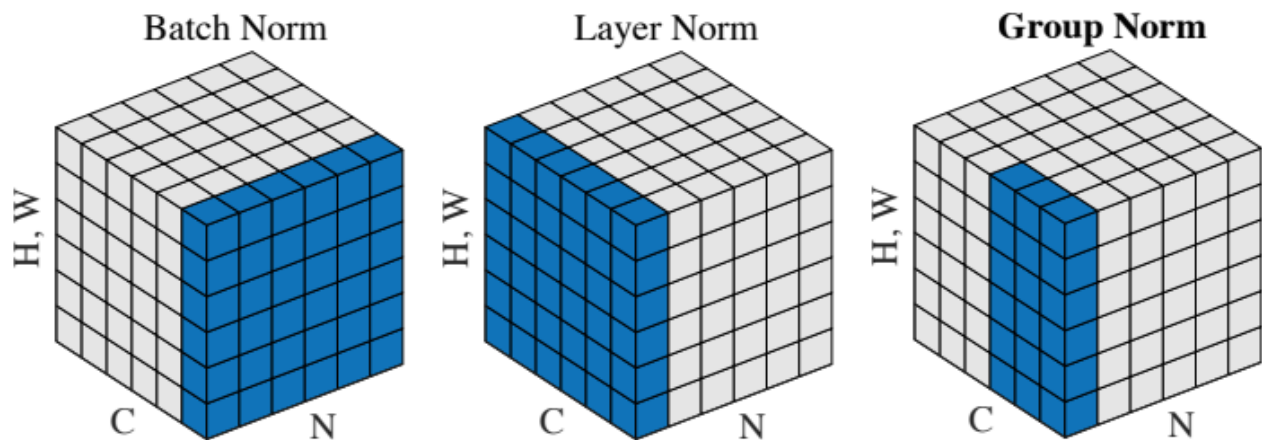
dx error:  2.786648197756335e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

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/>).

Group normalization: forward

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

```
In [5]: 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: [ 0.0353638 -0.0353638  0.03878463 -0.03878463]
Stds:  [1.05136914 0.94452101 1.02890674 0.96868012]
```


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 [16]: 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:  4.718196280640773e-08
dgamma error:  4.594418457582979e-12
dbeta error:  4.9347398870131864e-12
```

What's this PyTorch 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, PyTorch (or TensorFlow, if you switch over to that notebook).

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- 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.

PyTorch versions

This notebook assumes that you are using **PyTorch version 0.4**. Prior to this version, Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 0.4 also separates a Tensor's datatype from its device, and uses numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

How will I learn PyTorch?

Justin Johnson has made an excellent [tutorial \(https://github.com/jcjohnson/pytorch-examples\)](https://github.com/jcjohnson/pytorch-examples) for PyTorch.

You can also find the detailed [API doc \(http://pytorch.org/docs/stable/index.html\)](http://pytorch.org/docs/stable/index.html) here. If you have other questions that are not addressed by the API docs, the [PyTorch forum \(https://discuss.pytorch.org/\)](https://discuss.pytorch.org/) is a much better place to ask than StackOverflow.

Table of Contents

This assignment has 5 parts. You will learn PyTorch on different levels of abstractions, which will help you understand it better and prepare you for the final project.

1. Preparation: we will use CIFAR-10 dataset.
2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use `nn.Module` to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use `nn.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
<code>nn.Module</code>	High	Medium
<code>nn.Sequential</code>	Low	High

Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

```
In [1]: import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np
```

```

In [2]: NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
    T.ToTensor(),
    T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader
# which iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                       sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                             transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)

Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified

```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode.

The global variables `dtype` and `device` will control the data types throughout this assignment.

```
In [3]: USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)

using device: cpu
```

Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if `x` is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of `x` with respect to the scalar loss at the end.

PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape $N \times C \times H \times W$, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the width of the intermediate feature map in pixels

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 $C \times H \times W$ values per representation into a single long vector. The `flatten` function below first reads in the N , C , H , and W values 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 $N \times ??$, where $??$ is allowed to be anything (in this case, it will be $C \times H \times W$, but we don't need to specify that explicitly).

```
In [10]: def flatten(x):
          N = x.shape[0] # read in N, C, H, W
          return x.view(N, -1) # "flatten" the C * H * W values into a single
                                vector per image

          def test_flatten():
              x = torch.arange(12).view(2, 1, 3, 2)
              print('Before flattening: ', x)
              print('After flattening: ', flatten(x))

          test_flatten()

Before flattening:  tensor([[[[ 0.,  1.],
                               [ 2.,  3.],
                               [ 4.,  5.]]],
                           [[[ 6.,  7.],
                               [ 8.,  9.],
                               [10., 11.]]]])
After flattening:  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
                           [ 6.,  7.,  8.,  9., 10., 11.]])
```

Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.


```

In [11]: import torch.nn.functional as F # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass 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 PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch o
    f
        input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the
    network;
        w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification sco
    res for
        the input data x.
    """
    # first we flatten the image
    x = flatten(x) # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Sin
    ce w1 and
    # w2 have requires_grad=True, operations involving these Tensors wil
    l cause
    # PyTorch to build a computational graph, allowing automatic computa
    tion of
    # gradients. Since we are no longer implementing the backward pass b
    y hand we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x

def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype) # minibatch size 64, feature
    dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size()) # you should see [64, 10]

```

```
torch.nn.functional.conv2d(x, w, b, padding=2))
```

Barebones PyTorch: 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. Like above, we can immediately test our implementation by passing zeros through the 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: <http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d> (<http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d>); pay attention to the shapes of convolutional filters!

```

In [19]: def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW
        2) giving weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can you
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    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.
    #
    #####
    #####
    x = F.conv2d(x, conv_w1, bias=conv_b1, padding=2)
    x = F.relu(x)
    x = F.conv2d(x, conv_w2, bias=conv_b2, padding=1)
    x = F.relu(x)
    x = flatten(x)
    scores = x.mm(fc_w) + fc_b
    #####
    #####
    #
    #
    #
    #####
    #####
    return scores

```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

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

```
In [20]: def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype) # minibatch size 64,
    image size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype) # [out_channel, in
    _channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,)) # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype) # [out_channel, in
    _channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,)) # out_channel

    # you must calculate the shape of the tensor after two conv layers,
    before the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2,
    fc_w, fc_b])
    print(scores.size()) # you should see [64, 10]
    three_layer_convnet_test()

torch.Size([64, 10])
```

Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, <https://arxiv.org/abs/1502.01852> (<https://arxiv.org/abs/1502.01852>).

```
In [21]: def random_weight(shape):
        """
        Create random Tensors for weights; setting requires_grad=True means
        that we
        want to compute gradients for these Tensors during the backward pas
        s.
        We use Kaiming normalization: sqrt(2 / fan_in)
        """
        if len(shape) == 2: # FC weight
            fan_in = shape[0]
        else:
            fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_chann
            el, kH, kW]
        # randn is standard normal distribution generator.
        w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fa
            n_in)
        w.requires_grad = True
        return w

    def zero_weight(shape):
        return torch.zeros(shape, device=device, dtype=dtype, requires_grad=
            True)

    # create a weight of shape [3 x 5]
    # you should see the type `torch.cuda.FloatTensor` if you use GPU.
    # Otherwise it should be `torch.FloatTensor`
    random_weight((3, 5))
```

```
Out[21]: tensor([[ -0.3931, -1.2231, -1.2546, -0.2812,  0.4980],
                 [-0.2925, -1.6871, -0.5039, -1.3093, -0.6593],
                 [-1.4274,  0.0913, -2.6188,  0.1476,  0.4240]])
```

Barebones PyTorch: 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.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

```
In [22]: def check_accuracy_part2(loader, model_fn, params):
        """
        Check the accuracy of a classification model.

        Inputs:
        - loader: A DataLoader for the data split we want to check
        - model_fn: A function that performs the forward pass of the model,
          with the signature scores = model_fn(x, params)
        - params: List of PyTorch Tensors giving parameters of the model

        Returns: Nothing, but prints the accuracy of the model
        """
        split = 'val' if loader.dataset.train else 'test'
        print('Checking accuracy on the %s set' % split)
        num_correct, num_samples = 0, 0
        with torch.no_grad():
            for x, y in loader:
                x = x.to(device=device, dtype=dtype) # move to device, e.g.
GPU
                y = y.to(device=device, dtype=torch.int64)
                scores = model_fn(x, params)
                _, preds = scores.max(1)
                num_correct += (preds == y).sum()
                num_samples += preds.size(0)
            acc = float(num_correct) / num_samples
            print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples
, 100 * acc))
```

BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can [read about it here](http://pytorch.org/docs/stable/nn.html#cross-entropy) (<http://pytorch.org/docs/stable/nn.html#cross-entropy>).

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

```

In [23]: def train_part2(model_fn, params, learning_rate):
        """
        Train a model on CIFAR-10.

        Inputs:
        - model_fn: A Python function that performs the forward pass of the
        model.
          It should have the signature scores = model_fn(x, params) where x
          is a
          PyTorch Tensor of image data, params is a list of PyTorch Tensors
          giving
          model weights, and scores is a PyTorch Tensor of shape (N, C) givi
          ng
          scores for the elements in x.
        - params: List of PyTorch Tensors giving weights for the model
        - learning_rate: Python scalar giving the learning rate to use for S
        GD

        Returns: Nothing
        """
        for t, (x, y) in enumerate(loader_train):
            # Move the data to the proper device (GPU or CPU)
            x = x.to(device=device, dtype=dtype)
            y = y.to(device=device, dtype=torch.long)

            # Forward pass: compute scores and loss
            scores = model_fn(x, params)
            loss = F.cross_entropy(scores, y)

            # Backward pass: PyTorch figures out which Tensors in the comput
            ational
            # graph has requires_grad=True and uses backpropagation to compu
            te the
            # gradient of the loss with respect to these Tensors, and stores
            the
            # gradients in the .grad attribute of each Tensor.
            loss.backward()

            # Update parameters. We don't want to backpropagate through the
            # parameter updates, so we scope the updates under a torch.no_gr
            ad()
            # context manager to prevent a computational graph from being bu
            ilt.

            with torch.no_grad():
                for w in params:
                    w -= learning_rate * w.grad

                    # Manually zero the gradients after running the backward
                    pass

                    w.grad.zero_()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part2(loader_val, model_fn, params)
                print()

```

BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, w_1 and w_2 .

Each minibatch of CIFAR has 64 examples, so the tensor shape is $[64, 3, 32, 32]$.

After flattening, x shape should be $[64, 3 * 32 * 32]$. This will be the size of the first dimension of w_1 . The second dimension of w_1 is the hidden layer size, which will also be the first dimension of w_2 .

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

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


```
In [24]: hidden_layer_size = 4000
         learning_rate = 1e-2

         w1 = random_weight((3 * 32 * 32, hidden_layer_size))
         w2 = random_weight((hidden_layer_size, 10))

         train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.7809
Checking accuracy on the val set
Got 145 / 1000 correct (14.50%)
```

```
Iteration 100, loss = 2.4663
Checking accuracy on the val set
Got 311 / 1000 correct (31.10%)
```

```
Iteration 200, loss = 1.6822
Checking accuracy on the val set
Got 359 / 1000 correct (35.90%)
```

```
Iteration 300, loss = 1.9992
Checking accuracy on the val set
Got 395 / 1000 correct (39.50%)
```

```
Iteration 400, loss = 1.9690
Checking accuracy on the val set
Got 421 / 1000 correct (42.10%)
```

```
Iteration 500, loss = 1.5699
Checking accuracy on the val set
Got 390 / 1000 correct (39.00%)
```

```
Iteration 600, loss = 1.8646
Checking accuracy on the val set
Got 445 / 1000 correct (44.50%)
```

```
Iteration 700, loss = 2.0149
Checking accuracy on the val set
Got 426 / 1000 correct (42.60%)
```

BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

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

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

```

In [26]: learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

#####
#####
# TODO: Initialize the parameters of a three-layer ConvNet.
#
#####
#####
# image size [3, 32, 32]

conv_w1 = random_weight((channel_1, 3, 5, 5)) # padding=2, stride=1, 32,
32, 32
conv_b1 = zero_weight((channel_1))

conv_w2 = random_weight((channel_2, channel_1, 3, 3)) # padding=1, stride=1, 16, 32, 32
conv_b2 = zero_weight((channel_2))

fc_w = random_weight((channel_2*32*32, 10))
fc_b = zero_weight(10)
#####
#####
#                                     END OF YOUR CODE
#
#####
#####

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)

```

Iteration 0, loss = 3.7716
Checking accuracy on the val set
Got 125 / 1000 correct (12.50%)

Iteration 100, loss = 2.0464
Checking accuracy on the val set
Got 331 / 1000 correct (33.10%)

Iteration 200, loss = 1.8633
Checking accuracy on the val set
Got 384 / 1000 correct (38.40%)

Iteration 300, loss = 1.6765
Checking accuracy on the val set
Got 406 / 1000 correct (40.60%)

Iteration 400, loss = 1.6158
Checking accuracy on the val set
Got 409 / 1000 correct (40.90%)

Iteration 500, loss = 1.4597
Checking accuracy on the val set
Got 429 / 1000 correct (42.90%)

Iteration 600, loss = 1.5981
Checking accuracy on the val set
Got 456 / 1000 correct (45.60%)

Iteration 700, loss = 1.5273
Checking accuracy on the val set
Got 461 / 1000 correct (46.10%)

Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the [doc \(http://pytorch.org/docs/master/optim.html\)](http://pytorch.org/docs/master/optim.html) for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the [doc \(http://pytorch.org/docs/master/nn.html\)](http://pytorch.org/docs/master/nn.html) to learn more about the dozens of builtin layers. **Warning:** don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

```
In [43]: class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype) # minibatch size 64,
    feature dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size()) # you should see [64, 10]
test_TwoLayerFC()

torch.Size([64, 10])
```

Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with channel₁ 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with channel₂ 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to num_{classes} classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: <http://pytorch.org/docs/stable/nn.html#conv2d> (<http://pytorch.org/docs/stable/nn.html#conv2d>).

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print (64 , 10) for the shape of the output scores.

```

In [45]: class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        #####
        # TODO: Set up the layers you need for a three-layer ConvNet with the
        # architecture defined above.
        #
        #####
        # assign layer objects to class attributes
        self.conv_1 = nn.Conv2d(in_channel, channel_1, 5, padding=2)
        nn.init.kaiming_normal_(self.conv_1.weight)

        self.conv_2 = nn.Conv2d(channel_1, channel_2, 3, padding=1)
        nn.init.kaiming_normal_(self.conv_2.weight)

        self.relu = nn.ReLU()

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        #####
        #
        #                                     END OF YOUR CODE
        #
        #####

    def forward(self, x):
        scores = None
        #####
        # TODO: Implement the forward function for a 3-layer ConvNet. you
        # should use the layers you defined in __init__ and specify the
        # connectivity of those layers in forward()
        #
        #####
        conv_1_out = self.relu(self.conv_1(x))
        conv_2_out = self.relu(self.conv_2(conv_1_out))
        fc_out = self.fc(flatten(conv_2_out))
        scores=fc_out
        #####
        #
        #                                     END OF YOUR CODE
        #
        #####
        return scores

def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype) # minibatch size 64,

```

```

image size [3, 32, 32]
model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
scores = model(x)
print(scores.size()) # you should see [64, 10]
test_ThreeLayerConvNet()
torch.Size([64, 10])

```

Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

```

In [4]: def check_accuracy_part34(loader, model):
        if loader.dataset.train:
            print('Checking accuracy on validation set')
        else:
            print('Checking accuracy on test set')
        num_correct = 0
        num_samples = 0
        model.eval() # set model to evaluation mode
        with torch.no_grad():
            for x, y in loader:
                x = x.to(device=device, dtype=dtype) # move to device, e.g.
GPU
                y = y.to(device=device, dtype=torch.long)
                scores = model(x)
                _, preds = scores.max(1)
                num_correct += (preds == y).sum()
                num_samples += preds.size(0)
            acc = float(num_correct) / num_samples
            print('Got %d / %d correct (%.2f)' % (num_correct, num_samples,
100 * acc))

```

Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.


```

In [7]: def train_part34(model, optimizer, epochs=1):
        """
        Train a model on CIFAR-10 using the PyTorch Module API.

        Inputs:
        - model: A PyTorch Module giving the model to train.
        - optimizer: An Optimizer object we will use to train the model
        - epochs: (Optional) A Python integer giving the number of epochs to
        train for

        Returns: Nothing, but prints model accuracies during training.
        """
        model = model.to(device=device) # move the model parameters to CPU/
GPU
        for e in range(epochs):
            for t, (x, y) in enumerate(loader_train):
                model.train() # put model to training mode
                x = x.to(device=device, dtype=dtype) # move to device, e.g.
GPU
                y = y.to(device=device, dtype=torch.long)

                scores = model(x)
                loss = F.cross_entropy(scores, y)

                # Zero out all of the gradients for the variables which the
optimizer
                # will update.
                optimizer.zero_grad()

                # This is the backwards pass: compute the gradient of the lo
ss with
                # respect to each parameter of the model.
                loss.backward()

                # Actually update the parameters of the model using the grad
ients
                # computed by the backwards pass.
                optimizer.step()

                if t % print_every == 0:
                    print('Iteration %d, loss = %.4f' % (t, loss.item()))
                    check_accuracy_part34(loader_val, model)
                    print()

```

Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

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

```
In [37]: hidden_layer_size = 4000
         learning_rate = 1e-2
         model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
         optimizer = optim.SGD(model.parameters(), lr=learning_rate)

         train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.9163
Checking accuracy on validation set
Got 156 / 1000 correct (15.60)
```

```
Iteration 100, loss = 1.9619
Checking accuracy on validation set
Got 328 / 1000 correct (32.80)
```

```
Iteration 200, loss = 2.2899
Checking accuracy on validation set
Got 395 / 1000 correct (39.50)
```

```
Iteration 300, loss = 1.4229
Checking accuracy on validation set
Got 418 / 1000 correct (41.80)
```

```
Iteration 400, loss = 1.6534
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)
```

```
Iteration 500, loss = 1.7735
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)
```

```
Iteration 600, loss = 2.0504
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)
```

```
Iteration 700, loss = 1.6963
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)
```

Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

```

In [47]: learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
#####
#####
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
#####
#####
model = ThreeLayerConvNet(in_channel=3, channel_1=channel_1, channel_2=channel_2, num_classes=10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
#####
#####
#                                     END OF YOUR CODE

#####
#####

train_part34(model, optimizer)

Iteration 0, loss = 3.4417
Checking accuracy on validation set
Got 136 / 1000 correct (13.60)

Iteration 100, loss = 1.9838
Checking accuracy on validation set
Got 349 / 1000 correct (34.90)

Iteration 200, loss = 1.7289
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 300, loss = 1.6309
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Iteration 400, loss = 1.5151
Checking accuracy on validation set
Got 448 / 1000 correct (44.80)

Iteration 500, loss = 1.7477
Checking accuracy on validation set
Got 472 / 1000 correct (47.20)

Iteration 600, loss = 1.6734
Checking accuracy on validation set
Got 496 / 1000 correct (49.60)

Iteration 700, loss = 1.2378
Checking accuracy on validation set
Got 478 / 1000 correct (47.80)

```

Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

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

```
In [48]: # We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                       momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2988
Checking accuracy on validation set
Got 158 / 1000 correct (15.80)
```

```
Iteration 100, loss = 1.9126
Checking accuracy on validation set
Got 419 / 1000 correct (41.90)
```

```
Iteration 200, loss = 1.6231
Checking accuracy on validation set
Got 412 / 1000 correct (41.20)
```

```
Iteration 300, loss = 1.5393
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)
```

```
Iteration 400, loss = 1.4317
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)
```

```
Iteration 500, loss = 1.7577
Checking accuracy on validation set
Got 416 / 1000 correct (41.60)
```

```
Iteration 600, loss = 1.4025
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)
```

```
Iteration 700, loss = 1.7251
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)
```

Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

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

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

```

In [54]: channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

#####
#####
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the
#
# Sequential API.
#
#####
#####
model = nn.Sequential(
    nn.Conv2d(3, channel_1, 5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, 3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                        momentum=0.9, nesterov=True)
#####
#####
#
#
#####
#####

train_part34(model, optimizer)

```



```
Iteration 0, loss = 2.3068  
Checking accuracy on validation set  
Got 130 / 1000 correct (13.00)
```

```
Iteration 100, loss = 1.3418  
Checking accuracy on validation set  
Got 458 / 1000 correct (45.80)
```

```
Iteration 200, loss = 1.5857  
Checking accuracy on validation set  
Got 469 / 1000 correct (46.90)
```

```
Iteration 300, loss = 1.3015  
Checking accuracy on validation set  
Got 512 / 1000 correct (51.20)
```

```
Iteration 400, loss = 1.2726  
Checking accuracy on validation set  
Got 538 / 1000 correct (53.80)
```

```
Iteration 500, loss = 1.3085  
Checking accuracy on validation set  
Got 559 / 1000 correct (55.90)
```

```
Iteration 600, loss = 1.5306  
Checking accuracy on validation set  
Got 588 / 1000 correct (58.80)
```

```
Iteration 700, loss = 1.1805  
Checking accuracy on validation set  
Got 584 / 1000 correct (58.40)
```

Part V. CIFAR-10 open-ended challenge

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

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the `check_accuracy` and `train` functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: <http://pytorch.org/docs/stable/nn.html> (<http://pytorch.org/docs/stable/nn.html>)
- Activations: <http://pytorch.org/docs/stable/nn.html#non-linear-activations> (<http://pytorch.org/docs/stable/nn.html#non-linear-activations>)
- Loss functions: <http://pytorch.org/docs/stable/nn.html#loss-functions> (<http://pytorch.org/docs/stable/nn.html#loss-functions>)
- Optimizers: <http://pytorch.org/docs/stable/optim.html> (<http://pytorch.org/docs/stable/optim.html>)

Things you might try:

- **Filter size:** Above we used 5x5; would smaller filters be more efficient?
- **Number of filters:** Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution:** Do you use max pooling or just stride convolutions?
- **Batch normalization:** Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture:** The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
 - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
 - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
 - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling:** Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1, Filter#), which is then reshaped into a (Filter#) vector. This is used in [Google's Inception Network](https://arxiv.org/abs/1512.00567) (<https://arxiv.org/abs/1512.00567>) (See Table 1 for their architecture).
- **Regularization:** Add l2 weight regularization, or perhaps use Dropout.

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.

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 (<https://arxiv.org/abs/1512.03385>) where the input from the previous layer is added to the output.
 - DenseNets (<https://arxiv.org/abs/1608.06993>) where inputs into previous layers are concatenated together.
 - This blog has an in-depth overview (<https://chatbotlife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32>).

```

In [12]: #####
#####
# TODO:
#
# Experiment with any architectures, optimizers, and hyperparameters.
#
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epoch
s.      #
#
#
# Note that you can use the check_accuracy function to evaluate on eithe
r      #
# the test set or the validation set, by passing either loader_test or
#
# loader_val as the second argument to check_accuracy. You should not to
uch    #
# the test set until you have finished your architecture and hyperparam
eter   #
# tuning, and only run the test set once at the end to report a final va
lue.   #
#####
#####
model = None
optimizer = None

learning_rate = 1e-3

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=
stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion*planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion*planes, kernel_size=
1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion*planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):

```

```

    super(ResNet, self).__init__()
    self.in_planes = 64

    self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1
, bias=False) # 3,32,32 -> 64, 32, 32
    self.bn1 = nn.BatchNorm2d(64) # 64, 32, 32 -> 64,32,32
    self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=
1)
    self.layer2 = self._make_layer(block, 128, num_blocks[1], stride
=2)
    self.layer3 = self._make_layer(block, 256, num_blocks[2], stride
=2)
    self.layer4 = self._make_layer(block, 512, num_blocks[3], stride
=2)
    self.linear = nn.Linear(512*block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        # planes = 128, num_blocks = 2, stride = 2
        strides = [stride] + [1]*(num_blocks-1) # strides = [2, 1]
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride)) # (in_p
lanes=64, planes=128, stride=2)
            self.in_planes = planes * block.expansion #
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out

def ResNet18():
    return ResNet(BasicBlock, [2,2,2,2])

model = ResNet18()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
#####
#####
#
END OF YOUR CODE

#####
#####

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)

```

Iteration 0, loss = 2.3030
Checking accuracy on validation set
Got 113 / 1000 correct (11.30)

Iteration 100, loss = 1.7599
Checking accuracy on validation set
Got 337 / 1000 correct (33.70)

Iteration 200, loss = 1.2863
Checking accuracy on validation set
Got 430 / 1000 correct (43.00)

Iteration 300, loss = 1.2473
Checking accuracy on validation set
Got 481 / 1000 correct (48.10)

Iteration 400, loss = 1.1588
Checking accuracy on validation set
Got 561 / 1000 correct (56.10)

Iteration 500, loss = 0.8924
Checking accuracy on validation set
Got 546 / 1000 correct (54.60)

Iteration 600, loss = 0.9706
Checking accuracy on validation set
Got 561 / 1000 correct (56.10)

Iteration 700, loss = 1.1106
Checking accuracy on validation set
Got 641 / 1000 correct (64.10)

Iteration 0, loss = 1.0607
Checking accuracy on validation set
Got 662 / 1000 correct (66.20)

Iteration 100, loss = 0.9488
Checking accuracy on validation set
Got 644 / 1000 correct (64.40)

Iteration 200, loss = 0.6660
Checking accuracy on validation set
Got 654 / 1000 correct (65.40)

Iteration 300, loss = 0.7685
Checking accuracy on validation set
Got 670 / 1000 correct (67.00)

Iteration 400, loss = 1.1444
Checking accuracy on validation set
Got 683 / 1000 correct (68.30)

Iteration 500, loss = 0.7827
Checking accuracy on validation set
Got 736 / 1000 correct (73.60)

Iteration 600, loss = 0.8620

```
Checking accuracy on validation set  
Got 680 / 1000 correct (68.00)
```

```
Iteration 700, loss = 0.9091  
Checking accuracy on validation set  
Got 705 / 1000 correct (70.50)
```

```

-----
----
KeyboardInterrupt                                Traceback (most recent call l
ast)
<ipython-input-12-508f38d1e4a4> in <module>()
    82
    83 # You should get at least 70% accuracy
----> 84 train_part34(model, optimizer, epochs=10)

<ipython-input-7-0807bf043cab> in train_part34(model, optimizer, epoch
s)
    17         y = y.to(device=device, dtype=torch.long)
    18
----> 19         scores = model(x)
    20         loss = F.cross_entropy(scores, y)
    21

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/mod
ule.py in __call__(self, *input, **kwargs)
    489         result = self._slow_forward(*input, **kwargs)
    490     else:
--> 491         result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)

<ipython-input-12-508f38d1e4a4> in forward(self, x)
    66         out = self.layer2(out)
    67         out = self.layer3(out)
----> 68         out = self.layer4(out)
    69         out = F.avg_pool2d(out, 4)
    70         out = out.view(out.size(0), -1)

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/mod
ule.py in __call__(self, *input, **kwargs)
    489         result = self._slow_forward(*input, **kwargs)
    490     else:
--> 491         result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/con
tainer.py in forward(self, input)
    89         def forward(self, input):
    90             for module in self._modules.values():
----> 91                 input = module(input)
    92             return input
    93

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/mod
ule.py in __call__(self, *input, **kwargs)
    489         result = self._slow_forward(*input, **kwargs)
    490     else:
--> 491         result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)

<ipython-input-12-508f38d1e4a4> in forward(self, x)

```



```

33
34     def forward(self, x):
--> 35         out = F.relu(self.bn1(self.conv1(x)))
36         out = self.bn2(self.conv2(out))
37         out += self.shortcut(x)

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/mod
ule.py in __call__(self, *input, **kwargs)
489         result = self._slow_forward(*input, **kwargs)
490     else:
--> 491         result = self.forward(*input, **kwargs)
492     for hook in self._forward_hooks.values():
493         hook_result = hook(self, input, result)

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/modules/bat
chnorm.py in forward(self, input)
47         return F.batch_norm(
48             input, self.running_mean, self.running_var, self.we
ight, self.bias,
--> 49             self.training or not self.track_running_stats, sel
f.momentum, self.eps)
50
51     def extra_repr(self):

/home/shared/anaconda3/lib/python3.6/site-packages/torch/nn/functional.
py in batch_norm(input, running_mean, running_var, weight, bias, traini
ng, momentum, eps)
1192     return torch.batch_norm(
1193         input, weight, bias, running_mean, running_var,
-> 1194         training, momentum, eps, torch.backends.cudnn.enabled
1195     )
1196

```

KeyboardInterrupt:

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: Describe what you did

- Implement ResNet18 for CIFAR-10 dataset, refer to <https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L96> (<https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L96>), Core logic is $\text{res_out} = \text{conv2d}(X) + X$, stretch X to meet the channel of $\text{Conv}(X)$ when necessary

Test set -- run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in `best_model`). Think about how this compares to your validation set accuracy.

```
In [13]: best_model = model  
         check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set  
Got 7492 / 10000 correct (74.92)
```