3 - ConvolutionalNetworks

March 14, 2019

1 Convolutional Networks

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

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

Acknowledgement: This exercise is adapted from Stanford CS231n.

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

import numpy as np
import matplotlib.pyplot as plt
from libs.classifiers.cnn import *
from libs.data_utils import get_CIFAR10_data
from libs.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from libs.layers import *
from libs.fast_layers import *
from libs.solver import Solver

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

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

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

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

data = get_CIFAR10_data()
```

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

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

2 Convolution: Naive forward pass

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

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

You can test your implementation by running the following:

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

        conv_param = {'stride': 2, 'pad': 1}
        out, _ = conv_forward_naive(x, w, b, conv_param)

        correct_out = np.array([[[[[-0.08759809, -0.10987781],
                                   [-0.18387192, -0.2109216 ]],
                                  [[ 0.21027089,  0.21661097],
                                   [ 0.22847626,  0.23004637]],
                                  [[ 0.50813986,  0.54309974],
                                   [ 0.64082444,  0.67101435]]],
                                 [[[-0.98053589, -1.03143541],
                                   [-1.19128892, -1.24695841]],
                                  [[ 0.69108355,  0.66880383],
                                   [ 0.59480972,  0.56776003]],
                                  [[ 2.36270298,  2.36904306],
                                   [ 2.38090835,  2.38247847]]]]])

        # Compare your output to ours; difference should be around 1e-8
        print('Testing conv_forward_naive')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

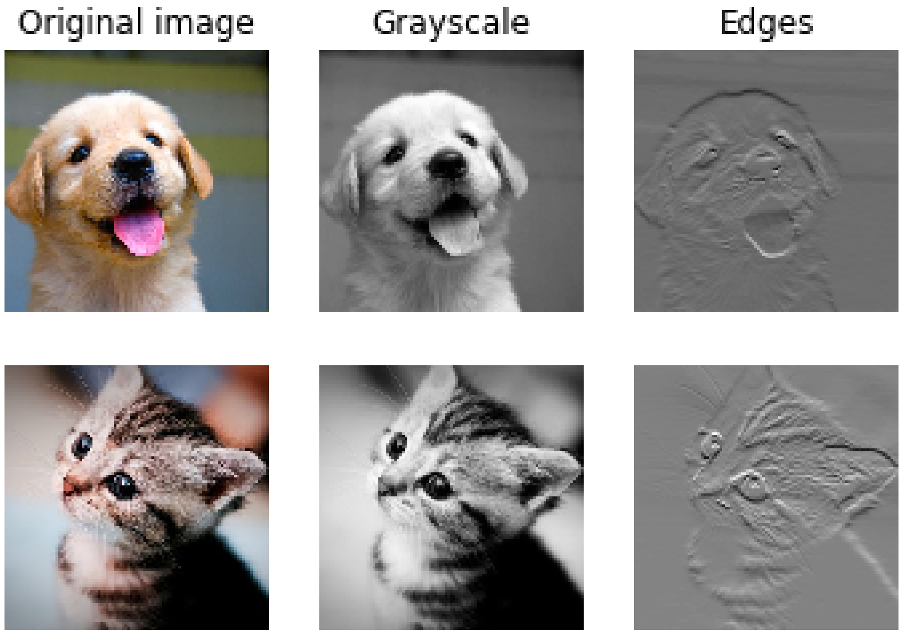
3 Aside: Image processing via convolutions

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

```
In [4]: from scipy.misc import imread, imresize

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

```python
# 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()
```

```
/anaconda3/envs/aienv/lib/python3.6/site-packages/ipykernel_launcher.py:3: DeprecationWarning:
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
/anaconda3/envs/aienv/lib/python3.6/site-packages/ipykernel_launcher.py:10: DeprecationWarning
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.3.0.
Use Pillow instead: ``numpy.array(Image.fromarray(arr).resize())``.
  # Remove the CWD from sys.path while we load stuff.
/anaconda3/envs/aienv/lib/python3.6/site-packages/ipykernel_launcher.py:11: DeprecationWarning
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.3.0.
Use Pillow instead: ``numpy.array(Image.fromarray(arr).resize())``.
  # This is added back by InteractiveShellApp.init_path()
```



4 Convolution: Naive backward pass

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

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

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

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

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

        # Your errors should be around 1e-9'
        print('Testing conv_backward_naive function')
        print('db error: ', rel_error(db, db_num))
```

```
print('dw error: ', rel_error(dw, dw_num))
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing conv_backward_naive function
db error:  1.0601537162848473e-11
dw error:  7.648956721713967e-11
dx error:  6.449319938664987e-09
```

5 Max pooling: Naive forward

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

Check your implementation by running the following:

```
In [6]: x_shape = (2, 3, 4, 4)
        x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
        pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

        out, _ = max_pool_forward_naive(x, pool_param)

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

        # Compare your output with ours. Difference should be around 1e-8.
        print('Testing max_pool_forward_naive function:')
        print('difference: ', rel_error(out, correct_out))
```

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

6 Max pooling: Naive backward

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

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

```
In [7]: x = np.random.randn(3, 2, 8, 8)
        dout = np.random.randn(3, 2, 4, 4)
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

        dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_param)

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

        # Your error should be around 1e-12
        print('Testing max_pool_backward_naive function:')
        print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.275634344353216e-12
```

7 Fast layers

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

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

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

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

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

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

```
In [8]: from libs.fast_layers import conv_forward_fast, conv_backward_fast
        from time import time

        # Reducing to lessen load on computer
        x = np.random.randn(100, 3, 9, 9)
        w = np.random.randn(25, 3, 3, 3)
        b = np.random.randn(25,)
        dout = np.random.randn(100, 25, 5, 5)
        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: 0.398419s
Fast: 0.006170s
Speedup: 64.573206x
Difference:  1.1569354904568532e-12

Testing conv_backward_fast:
Naive: 92.563465s
Fast: 0.002916s
Speedup: 31742.237920x
dx difference:  1.106225888839249e-12
dw difference:  3.1371133732211265e-13
db difference:  3.96077540473357e-15
```

In [9]:
```
from libs.fast_layers import max_pool_forward_fast, max_pool_backward_fast

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('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.305430s
fast: 0.003110s
speedup: 98.203603x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.332742s
speedup: 130.712639x
dx difference:  0.0
```

8 Convolutional "sandwich" layers

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

```
In [10]: from libs.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

          x = np.random.randn(2, 3, 16, 16)
          w = np.random.randn(3, 3, 3, 3)
          b = np.random.randn(3,)
          dout = np.random.randn(2, 3, 8, 8)
          conv_param = {'stride': 1, 'pad': 1}
```

```
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, conv_
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_

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:  3.0976634179344225e-07
dw error:  1.3799416958638531e-09
db error:  2.9077352162083648e-11
```

In [11]:
```
from libs.layer_utils import conv_relu_forward, conv_relu_backward

x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

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

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

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:  4.3184508188670987e-07
dw error:  4.9801702570447816e-09
db error:  1.6153738932002093e-11
```

9 Three-layer ConvNet

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

Open the file `libs/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Run the following cells to help you debug:

9.1 Sanity check loss

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

```
In [12]: 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)
         print('Should be near: ', np.log(10))

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

```
Initial loss (no regularization):  2.3025858199418425
Should be near:  2.302585092994046
Initial loss (with regularization):  2.509066762491958
```

9.2 Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

```
In [13]: num_inputs = 2
         input_dim = (3, 16, 16)
         reg = 0.0
         num_classes = 10
         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)
```

```
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=Fals
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[p
```

```
W1 max relative error: 1.098667e-03
W2 max relative error: 3.391480e-02
W3 max relative error: 1.809480e-05
b1 max relative error: 1.761282e-04
b2 max relative error: 6.263619e-07
b3 max relative error: 1.278287e-09
```

9.3 Overfit small data

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

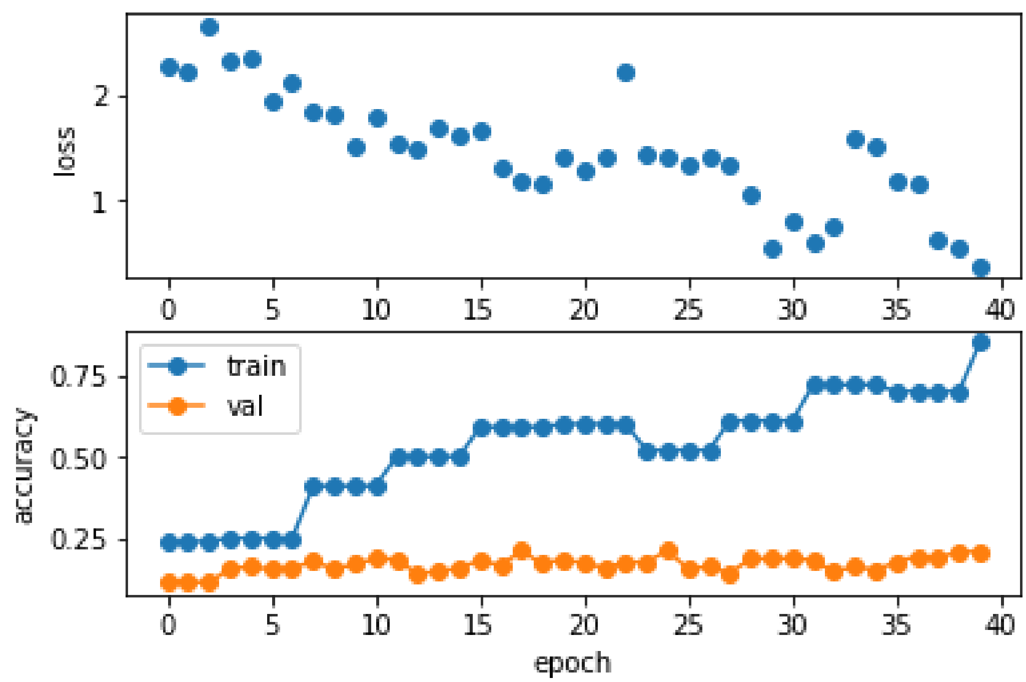
In [15]:
```
num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
                num_epochs=10, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 5e-3,
                },
                verbose=True, print_every=4)
solver.train()
```

```
(Epoch 0 / 10) (Iteration 1 / 40) loss: 2.264084 train acc: 0.240000 val_acc: 0.119000
(Epoch 1 / 10) (Iteration 5 / 40) loss: 2.359271 train acc: 0.250000 val_acc: 0.162000
(Epoch 2 / 10) (Iteration 9 / 40) loss: 1.823537 train acc: 0.410000 val_acc: 0.155000
(Epoch 3 / 10) (Iteration 13 / 40) loss: 1.479577 train acc: 0.500000 val_acc: 0.140000
(Epoch 4 / 10) (Iteration 17 / 40) loss: 1.312175 train acc: 0.590000 val_acc: 0.166000
(Epoch 5 / 10) (Iteration 21 / 40) loss: 1.274080 train acc: 0.600000 val_acc: 0.172000
(Epoch 6 / 10) (Iteration 25 / 40) loss: 1.420602 train acc: 0.520000 val_acc: 0.211000
(Epoch 7 / 10) (Iteration 29 / 40) loss: 1.044251 train acc: 0.610000 val_acc: 0.187000
(Epoch 8 / 10) (Iteration 33 / 40) loss: 0.764127 train acc: 0.720000 val_acc: 0.147000
```

```
(Epoch 9 / 10) (Iteration 37 / 40) loss: 1.144925 train acc: 0.700000 val_acc: 0.194000
```

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

```
In [16]: plt.subplot(2, 1, 1)
         plt.plot(solver.loss_history, 'o')
         plt.xlabel('iteration')
         plt.ylabel('loss')

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



9.4 Train the net

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

```
In [19]: model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.005)
```

```python
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()
```

```
(Epoch 0 / 1) (Iteration 1 / 980) loss: 2.312774 train acc: 0.102000 val_acc: 0.078000
(Epoch 0 / 1) (Iteration 21 / 980) loss: 2.208834 train acc: 0.102000 val_acc: 0.255000
(Epoch 0 / 1) (Iteration 41 / 980) loss: 1.970535 train acc: 0.102000 val_acc: 0.321000
(Epoch 0 / 1) (Iteration 61 / 980) loss: 1.769237 train acc: 0.102000 val_acc: 0.354000
(Epoch 0 / 1) (Iteration 81 / 980) loss: 2.203918 train acc: 0.102000 val_acc: 0.372000
(Epoch 0 / 1) (Iteration 101 / 980) loss: 1.969835 train acc: 0.102000 val_acc: 0.344000
(Epoch 0 / 1) (Iteration 121 / 980) loss: 1.957917 train acc: 0.102000 val_acc: 0.361000
(Epoch 0 / 1) (Iteration 141 / 980) loss: 1.724123 train acc: 0.102000 val_acc: 0.375000
(Epoch 0 / 1) (Iteration 161 / 980) loss: 2.154189 train acc: 0.102000 val_acc: 0.416000
(Epoch 0 / 1) (Iteration 181 / 980) loss: 1.716081 train acc: 0.102000 val_acc: 0.390000
(Epoch 0 / 1) (Iteration 201 / 980) loss: 2.000575 train acc: 0.102000 val_acc: 0.413000
(Epoch 0 / 1) (Iteration 221 / 980) loss: 1.783172 train acc: 0.102000 val_acc: 0.411000
(Epoch 0 / 1) (Iteration 241 / 980) loss: 1.976374 train acc: 0.102000 val_acc: 0.351000
(Epoch 0 / 1) (Iteration 261 / 980) loss: 1.582185 train acc: 0.102000 val_acc: 0.348000
(Epoch 0 / 1) (Iteration 281 / 980) loss: 1.648631 train acc: 0.102000 val_acc: 0.413000
(Epoch 0 / 1) (Iteration 301 / 980) loss: 1.622802 train acc: 0.102000 val_acc: 0.388000
(Epoch 0 / 1) (Iteration 321 / 980) loss: 1.961782 train acc: 0.102000 val_acc: 0.445000
(Epoch 0 / 1) (Iteration 341 / 980) loss: 1.674939 train acc: 0.102000 val_acc: 0.378000
(Epoch 0 / 1) (Iteration 361 / 980) loss: 1.607289 train acc: 0.102000 val_acc: 0.454000
(Epoch 0 / 1) (Iteration 381 / 980) loss: 1.743716 train acc: 0.102000 val_acc: 0.414000
(Epoch 0 / 1) (Iteration 401 / 980) loss: 1.713242 train acc: 0.102000 val_acc: 0.403000
(Epoch 0 / 1) (Iteration 421 / 980) loss: 1.945983 train acc: 0.102000 val_acc: 0.448000
(Epoch 0 / 1) (Iteration 441 / 980) loss: 1.749760 train acc: 0.102000 val_acc: 0.446000
(Epoch 0 / 1) (Iteration 461 / 980) loss: 1.689935 train acc: 0.102000 val_acc: 0.426000
(Epoch 0 / 1) (Iteration 481 / 980) loss: 1.840547 train acc: 0.102000 val_acc: 0.433000
(Epoch 0 / 1) (Iteration 501 / 980) loss: 1.423356 train acc: 0.102000 val_acc: 0.454000
(Epoch 0 / 1) (Iteration 521 / 980) loss: 1.267752 train acc: 0.102000 val_acc: 0.431000
(Epoch 0 / 1) (Iteration 541 / 980) loss: 1.846943 train acc: 0.102000 val_acc: 0.439000
(Epoch 0 / 1) (Iteration 561 / 980) loss: 1.868596 train acc: 0.102000 val_acc: 0.482000
(Epoch 0 / 1) (Iteration 581 / 980) loss: 1.439365 train acc: 0.102000 val_acc: 0.460000
(Epoch 0 / 1) (Iteration 601 / 980) loss: 1.580134 train acc: 0.102000 val_acc: 0.422000
(Epoch 0 / 1) (Iteration 621 / 980) loss: 1.549202 train acc: 0.102000 val_acc: 0.419000
(Epoch 0 / 1) (Iteration 641 / 980) loss: 1.373486 train acc: 0.102000 val_acc: 0.425000
(Epoch 0 / 1) (Iteration 661 / 980) loss: 1.919476 train acc: 0.102000 val_acc: 0.487000
(Epoch 0 / 1) (Iteration 681 / 980) loss: 1.731552 train acc: 0.102000 val_acc: 0.439000
(Epoch 0 / 1) (Iteration 701 / 980) loss: 1.824866 train acc: 0.102000 val_acc: 0.441000
(Epoch 0 / 1) (Iteration 721 / 980) loss: 1.555097 train acc: 0.102000 val_acc: 0.466000
(Epoch 0 / 1) (Iteration 741 / 980) loss: 1.675195 train acc: 0.102000 val_acc: 0.434000
(Epoch 0 / 1) (Iteration 761 / 980) loss: 1.785995 train acc: 0.102000 val_acc: 0.460000
```

```
(Epoch 0 / 1) (Iteration 781 / 980) loss: 1.802104 train acc: 0.102000 val_acc: 0.451000
(Epoch 0 / 1) (Iteration 801 / 980) loss: 1.421889 train acc: 0.102000 val_acc: 0.427000
(Epoch 0 / 1) (Iteration 821 / 980) loss: 1.724522 train acc: 0.102000 val_acc: 0.478000
(Epoch 0 / 1) (Iteration 841 / 980) loss: 1.517059 train acc: 0.102000 val_acc: 0.449000
(Epoch 0 / 1) (Iteration 861 / 980) loss: 1.540298 train acc: 0.102000 val_acc: 0.473000
(Epoch 0 / 1) (Iteration 881 / 980) loss: 1.685802 train acc: 0.102000 val_acc: 0.457000
(Epoch 0 / 1) (Iteration 901 / 980) loss: 1.387677 train acc: 0.102000 val_acc: 0.482000
(Epoch 0 / 1) (Iteration 921 / 980) loss: 1.883926 train acc: 0.102000 val_acc: 0.478000
(Epoch 0 / 1) (Iteration 941 / 980) loss: 1.888729 train acc: 0.102000 val_acc: 0.485000
(Epoch 0 / 1) (Iteration 961 / 980) loss: 1.522582 train acc: 0.102000 val_acc: 0.476000
```

```
In [20]: solver.check_accuracy(data['X_train'], data['y_train'])
```

```
Out[20]: 0.48391836734693877
```

9.5 Visualize Filters

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

```
In [21]: from libs.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()
```