ConvolutionalNetworks

October 28, 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.

[1]:
```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from ie590.classifiers.cnn import *
from ie590.data_utils import get_CIFAR10_data
from ie590.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from ie590.layers import *
from ie590.fast_layers import *
from ie590.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))))
```

[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 `ie590/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:

[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 2e-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

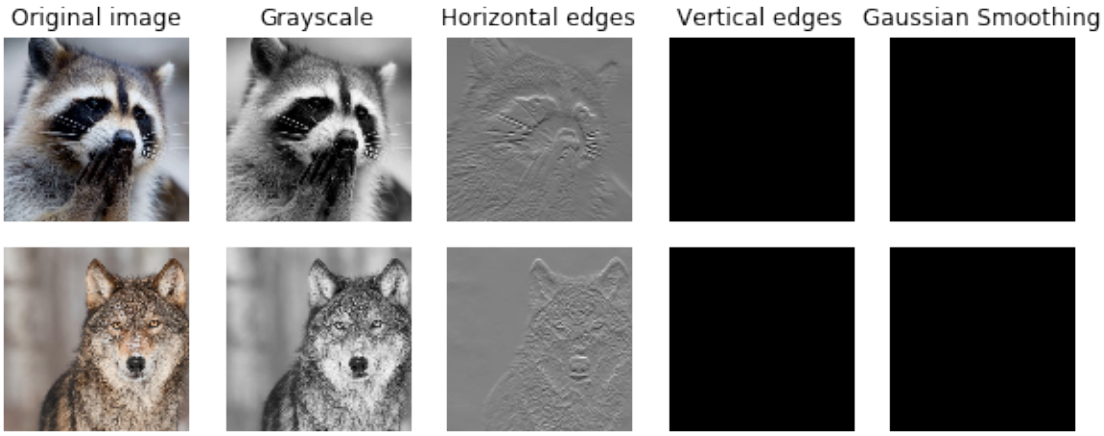
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection).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

[4]:
```
from imageio import imread
from PIL import Image

wolf = imread('notebook_images/wolf.jpg')
racoon = imread('notebook_images/racoon.jpg')
# wolf is wide, and racoon is already square
d = wolf.shape[1] - wolf.shape[0]
wolf_cropped = wolf[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_racoon = np.array(Image.fromarray(racoon).resize((img_size, img_size)))
resized_wolf = np.array(Image.fromarray(wolf_cropped).resize((img_size,␣
 ↪img_size)))
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resized_racoon.transpose((2, 0, 1))
x[1, :, :, :] = resized_wolf.transpose((2, 0, 1))

# Set up a convolutional weights holding 4 filters, each 3x3
w = np.zeros((4, 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]]

################################################################################
# TODO:                                                                        #
# Apply the following two filters: vertical edge, and Gaussian smoothing filter#
# (3x3) to the given racoon and wolf pictures. What you need to do is to modify#
# the given 3x3 matrices.                                                      #
# - Third filter detects vertical edges in the G(green) channel.               #
# - Fourth filter applies the Gausssian smoothing in the R(red) channel.       #
################################################################################
# Third filter.
w[2, 0, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
w[2, 1, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
w[2, 2, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

# Fourth filter.
w[3, 0, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
w[3, 1, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
w[3, 2, :, :] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
################################################################################
#                                END OF YOUR CODE                              #
################################################################################


# 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, 128, 0])

# 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_no_ax(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.figure(figsize=(10,4))
plt.subplots_adjust(hspace = .001)

plt.subplot(2, 5, 1)
imshow_no_ax(racoon, normalize=False)
plt.title('Original image')
plt.subplot(2, 5, 2)
imshow_no_ax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 5, 3)
imshow_no_ax(out[0, 1])
plt.title('Horizontal edges')
plt.subplot(2, 5, 4)
imshow_no_ax(out[0, 2])
plt.title('Vertical edges')
plt.subplot(2, 5, 5)
imshow_no_ax(out[0, 3])
```

```
plt.title('Gaussian Smoothing')

plt.subplot(2, 5, 6)
imshow_no_ax(wolf_cropped, normalize=False)
plt.subplot(2, 5, 7)
imshow_no_ax(out[1, 0])
plt.subplot(2, 5, 8)
imshow_no_ax(out[1, 1])
plt.subplot(2, 5, 9)
imshow_no_ax(out[1, 2])
plt.subplot(2, 5, 10)
imshow_no_ax(out[1, 3])

plt.show()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:64:
RuntimeWarning: invalid value encountered in true_divide
```



4 Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `ie590/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.

[5]:
```
np.random.seed(123)
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:  6.651568310721759e-10
dw error:  7.210432313808002e-11
db error:  7.469395900547805e-12
```

5 Max-Pooling: Naive forward

The given `max_pool_forward_naive` function in the file `ie590/layers.py` is the forward pass for the max-pooling operation. Please check the given implementation carefully and run the following:

[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       ]]]])
```

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

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

6 Max-Pooling: Naive backward

The given `max_pool_backward_naive` function in the file `ie590/layers.py` is the forward pass for the max-pooling operation. Please check the given implementation carefully and run the following:

[7]:
```
np.random.seed(123)
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)

# The 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:  3.2756292521229896e-12
```

7 Average-Pooling: Naive forward

This time, implement the forward pass for the average-pooling operation in the function `avg_pool_forward_naive` in the file `ie590/layers.py` yourself. Don't worry too much about computational efficiency.

Check your implementation by running the following:

[8]:
```
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, _ = avg_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.28157895, -0.26684211],
                          [-0.22263158, -0.20789474]],
```

```
                        [[-0.16368421, -0.14894737],
                         [-0.10473684, -0.09      ]],
                        [[-0.04578947, -0.03105263],
                         [ 0.01315789,  0.02789474]]],
                       [[[ 0.07210526,  0.08684211],
                         [ 0.13105263,  0.14578947]],
                        [[ 0.19      ,  0.20473684],
                         [ 0.24894737,  0.26368421]],
                        [[ 0.30789474,  0.32263158],
                         [ 0.36684211,  0.38157895]]]])

# The output should be on the order of e-7.
print('Testing avg_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing avg_pool_forward_naive function:
difference:  1.8000003183688645e-07
```

8 Average-Pooling: Naive backward

Implement the backward pass for the average-pooling operation in the function `avg_pool_backward_naive` in the file `ie590/layers.py`. You don't need to worry about computational efficiency.

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

[9]:
```
np.random.seed(123)
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: avg_pool_forward_naive(x,␣
 ↪pool_param)[0], x, dout)
out, cache = avg_pool_forward_naive(x, pool_param)
dx = avg_pool_backward_naive(dout, cache)

# The error should be on the order of e-11
print('Testing avg_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing avg_pool_backward_naive function:
dx error:  2.5480046567755598e-11
```

Inline Question #1: If you want to reduce the size of the image from $W \times H \times N$ to $W/4 \times H/4 \times N$, what should be stride, pool height and pool width?

Your Answer: *stride = 4, pool height* $HH = 4 + 2P_H$, *pool width* $WW = 4 + 2P\_W$, *where* P_H *and* P_W *are the amount of symmetric padding on both sides of height and width, respectively.*

Inline Question #2: Does max pooling act as a non-linear activation similar to the Sigmoid and tanh? If so, why do we need a ReLU right after the max pooling layer?

Your Answer: *Yes, max pooling is similar to a non-linear activation, but it only chooses a local maximum and lacks of many good features in classic activation layer such as zero-center and pushing the score to an extrame. If we only use max pool, the input to the next layer will be very noisy with a large range. Besides, max pooling cannot selet out negative values, making it harder to find good loss function to define the performance.*

Inline Question #3: If average pooling has to be represented as the kernel, how does the kernel look like if the we reduce the image from size $M \times N \times K$ to $M/2 \times N/2 \times K$.

Your Answer: The conv kernel will be K filters with stride S=2. Considering no padding, each filter will have size 2×2 and all elements being $1/4$. All biases must be 0 to get the average.

9 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 `ie590/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `ie590` 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:

[10]:
```
# Rel errors should be around e-9 or less
from ie590.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(123)
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: 0.202351s
Fast: 0.016087s
Speedup: 12.578848x
Difference:  6.519338615553678e-12

Testing conv_backward_fast:
Naive: 0.564266s
Fast: 0.008661s
Speedup: 65.153531x
dx difference:  7.900679034850285e-12
dw difference:  6.042774204602018e-14
db difference:  0.0
```

[11]:
```
# Relative errors should be close to 0.0
from ie590.fast_layers import max_pool_forward_fast, max_pool_backward_fast
from time import time

np.random.seed(123)
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: 0.146694s
fast: 0.002367s
speedup: 61.986500x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.389532s
fast: 0.011802s
speedup: 33.005717x
dx difference:  0.0
```

10 Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `ie590/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check they're working.

[12]:
```
from ie590.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(123)
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:  5.2133198331397e-08
dw error:  1.8908186704069546e-09
db error:  1.9424627683247526e-11
```

[13]:
```
from ie590.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(123)
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:  1.7979021700661513e-08
dw error:  3.751047138262295e-09
db error:  1.2633908726265282e-11
```

11 Four-layer ConvNet

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

Open the file `ie590/classifiers/cnn.py` and complete the implementation of the `FourLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

11.1 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 the loss should go up slightly.

[14]:
```
model = FourLayerConvNet()

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):  2.3025850878441325
Initial loss (with regularization):  2.5113279448235026
```

11.2 Gradient check

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

[15]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
```

```
num_classes = 10
np.random.seed(123)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = FourLayerConvNet(num_filters=3, filter_size=3,
                         input_dim=input_dim, hidden_dim=7,
                         dtype=np.float64)
loss, grads = model.loss(X, y)
# Correct implementations may have relative errors up to the order of e-1,
# but if you got an error of 1.0 for 'b3', then please proceed. We will not␣
 ↪deduct points from that error.
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: 4.134720e-02
W2 max relative error: 4.096642e-02
W3 max relative error: 1.857769e-02
W4 max relative error: 1.561920e-02
b1 max relative error: 1.585612e-02
b2 max relative error: 2.037872e-01
b3 max relative error: 1.000000e+00
b4 max relative error: 1.556248e-09
```

11.3 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.

[16]:
```
np.random.seed(123)

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 = FourLayerConvNet(weight_scale=1e-2)
```

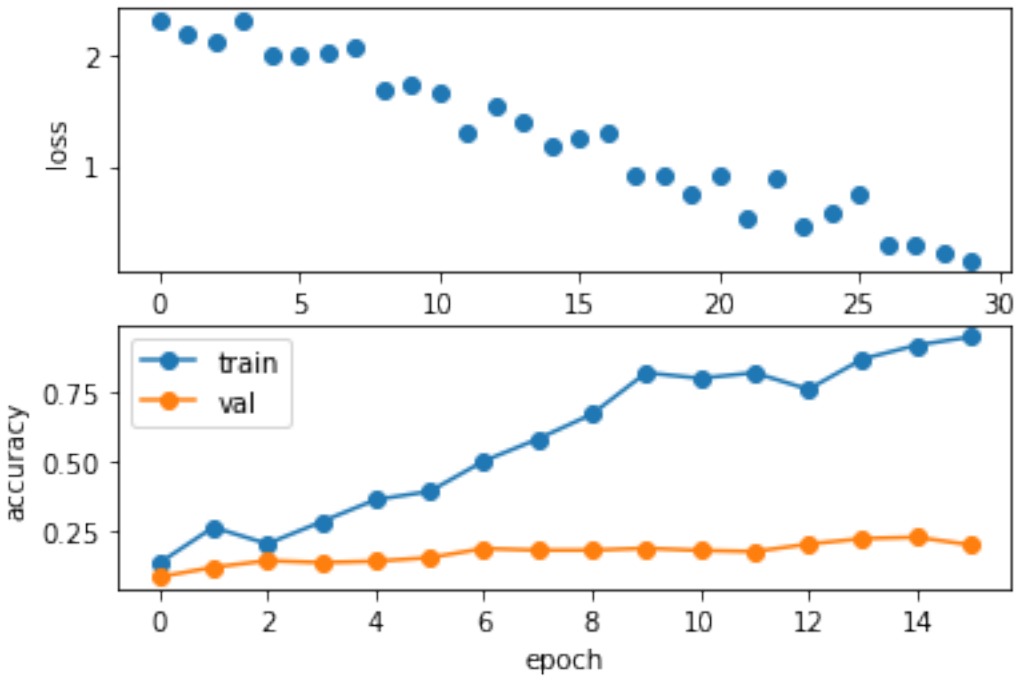
```
solver = Solver(model, small_data,
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.303732
(Epoch 0 / 15) train acc: 0.130000; val_acc: 0.080000
(Iteration 2 / 30) loss: 2.195160
(Epoch 1 / 15) train acc: 0.260000; val_acc: 0.115000
(Iteration 3 / 30) loss: 2.106891
(Iteration 4 / 30) loss: 2.314115
(Epoch 2 / 15) train acc: 0.200000; val_acc: 0.139000
(Iteration 5 / 30) loss: 1.999812
(Iteration 6 / 30) loss: 1.987454
(Epoch 3 / 15) train acc: 0.280000; val_acc: 0.132000
(Iteration 7 / 30) loss: 2.031552
(Iteration 8 / 30) loss: 2.077570
(Epoch 4 / 15) train acc: 0.360000; val_acc: 0.137000
(Iteration 9 / 30) loss: 1.678030
(Iteration 10 / 30) loss: 1.744371
(Epoch 5 / 15) train acc: 0.390000; val_acc: 0.150000
(Iteration 11 / 30) loss: 1.655414
(Iteration 12 / 30) loss: 1.301665
(Epoch 6 / 15) train acc: 0.500000; val_acc: 0.183000
(Iteration 13 / 30) loss: 1.538784
(Iteration 14 / 30) loss: 1.391864
(Epoch 7 / 15) train acc: 0.580000; val_acc: 0.177000
(Iteration 15 / 30) loss: 1.192692
(Iteration 16 / 30) loss: 1.245648
(Epoch 8 / 15) train acc: 0.670000; val_acc: 0.177000
(Iteration 17 / 30) loss: 1.305940
(Iteration 18 / 30) loss: 0.908247
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.184000
(Iteration 19 / 30) loss: 0.908391
(Iteration 20 / 30) loss: 0.744374
(Epoch 10 / 15) train acc: 0.800000; val_acc: 0.176000
(Iteration 21 / 30) loss: 0.909381
(Iteration 22 / 30) loss: 0.537355
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.172000
(Iteration 23 / 30) loss: 0.891799
(Iteration 24 / 30) loss: 0.450030
(Epoch 12 / 15) train acc: 0.760000; val_acc: 0.199000
(Iteration 25 / 30) loss: 0.574060
```

```
(Iteration 26 / 30) loss: 0.743062
(Epoch 13 / 15) train acc: 0.870000; val_acc: 0.219000
(Iteration 27 / 30) loss: 0.300220
(Iteration 28 / 30) loss: 0.290112
(Epoch 14 / 15) train acc: 0.920000; val_acc: 0.224000
(Iteration 29 / 30) loss: 0.226553
(Iteration 30 / 30) loss: 0.156236
(Epoch 15 / 15) train acc: 0.950000; val_acc: 0.196000
```

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

[17]:
```
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()
```



11.4 Train the net

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

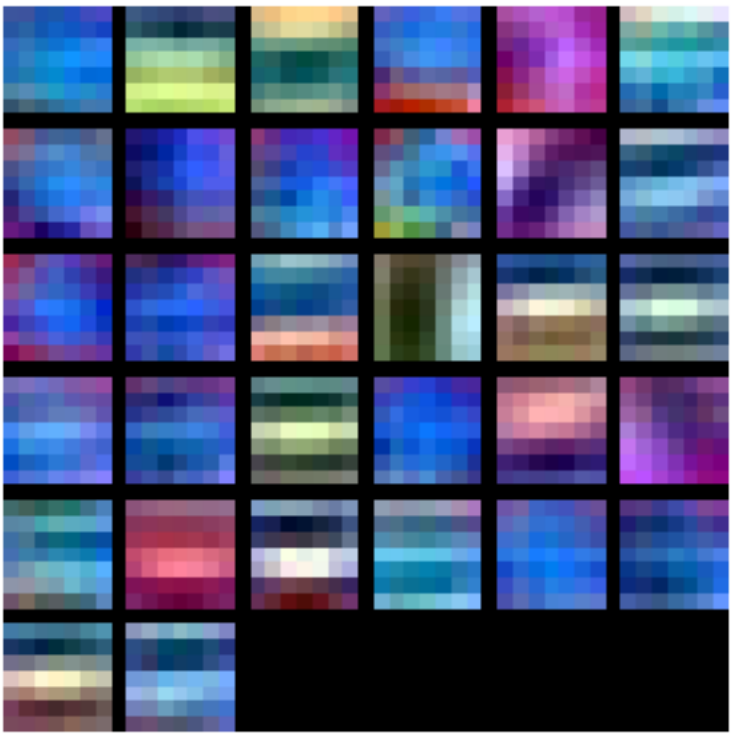
[18]:
```
model = FourLayerConvNet(weight_scale=0.001, hidden_dim=200, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=50)
solver.train()
```

```
(Iteration 1 / 490) loss: 2.303427
(Epoch 0 / 1) train acc: 0.105000; val_acc: 0.082000
(Iteration 51 / 490) loss: 2.302774
(Iteration 101 / 490) loss: 2.251280
(Iteration 151 / 490) loss: 2.136233
(Iteration 201 / 490) loss: 2.082164
(Iteration 251 / 490) loss: 1.918240
(Iteration 301 / 490) loss: 1.641616
(Iteration 351 / 490) loss: 1.508744
(Iteration 401 / 490) loss: 1.460834
(Iteration 451 / 490) loss: 1.443391
(Epoch 1 / 1) train acc: 0.438000; val_acc: 0.447000
```

11.5 Visualize Filters

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

[19]:
```
from ie590.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()
```



12 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 (link in `BatchNormalization.ipynb`), 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 every feature channel's statistics e.g. mean, variance to be relatively consistent both between different images, and different locations within the same image – after all, every feature channel is produced by the same convolutional filter! Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over the minibatch dimension `N` as well the spatial dimensions `H` and `W`.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

12.1 Spatial batch normalization: forward

In the file `ie590/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[20]:
```
np.random.seed(123)
# 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:  [ 9.79827084  9.5807009  10.81473958]
  Stds:  [4.47188275 4.28532851 4.92570937]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 1.15463195e-15  3.83026943e-16 -3.94129174e-16]
  Stds:  [0.99999975 0.99999973 0.99999979]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999925 3.99999891 4.99999897]

[autoreload of ie590.classifiers.cnn failed: Traceback (most recent call last):
  File "/opt/anaconda3/lib/python3.7/site-
```

```
packages/IPython/extensions/autoreload.py", line 245, in check
    superreload(m, reload, self.old_objects)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 450, in superreload
    update_generic(old_obj, new_obj)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 387, in update_generic
    update(a, b)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 357, in update_class
    update_instances(old, new)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 317, in update_instances
    update_instances(old, new, obj, visited)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 317, in update_instances
    update_instances(old, new, obj, visited)
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 317, in update_instances
    update_instances(old, new, obj, visited)
  [Previous line repeated 2 more times]
  File "/opt/anaconda3/lib/python3.7/site-
packages/IPython/extensions/autoreload.py", line 302, in update_instances
    visited.update({id(obj):obj})
MemoryError
]
```

[21]:
```python
np.random.seed(123)
# 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:  [ 0.04794133  0.04914931 -0.00719426  0.03869199]
  stds:  [1.03832542 1.01565838 0.99439152 1.02422673]
```

12.2 Spatial batch normalization: backward

In the file `ie590/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:

[22]:
```
np.random.seed(123)
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))
```

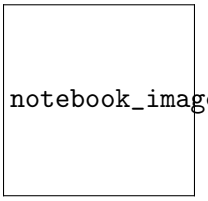
```
dx error:  1.1320575677049091e-08
dgamma error:  3.661038202496158e-12
dbeta error:  7.328805729901488e-12
```

13 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 [2] 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 [3] 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 [3])

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 [4]– 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 ECCV just in 2018 – this truly is still an ongoing and excitingly active field of research!

[2] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

[3] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

13.1 Group normalization: forward

In the file `ie590/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[23]:
```
np.random.seed(123)
# 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:  [10.37239844  9.75674244 10.30529237  9.39933631]
  Stds:  [4.76313353 4.41034782 3.22434302 3.6764582 ]
After spatial group normalization:
  Shape:  (1, 1, 1, 2, 6, 4, 5)
  Means:  [-4.68144042e-16  3.88578059e-16  1.25593980e-16 -5.62975592e-16]
  Stds:  [0.99999978 0.99999974 0.99999952 0.99999963]
```

13.2 Spatial group normalization: backward

In the file `ie590/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:

[24]:
```
np.random.seed(123)
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:  5.825038806802109e-07
dgamma error:  7.767479695082318e-10
dbeta error:  6.929114992872671e-11
```

[ ]: