Convolutional neural network layers

In this notebook, we will build the convolutional neural network layers. This will be followed by a spatial batchnorm, and then in the final notebook of this assignment, we will train a CNN to further improve the validation accuracy on CIFAR-10.

CS231n has built a solid API for building these modular frameworks and training them, and we will use their very well implemented framework as opposed to "reinventing the wheel." This includes using their Solver, various utility functions, their layer structure, and their implementation of fast CNN layers. This also includes nndl.fc_net, nndl.layers, and nndl.layer_utils. As in prior assignments, we thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu).

```
In [1]:  ## Import and setups

         import time
         import numpy as np
         import matplotlib.pyplot as plt
         from nndl.conv_layers import *
         from cs231n.data_utils import get_CIFAR10_data
         from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gr
         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-i
         %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))))
```

Implementing CNN layers

Just as we implemented modular layers for fully connected networks, batch normalization, and dropout, we'll want to implement modular layers for convolutional neural networks. These layers are in `nndl/conv_layers.py`.

Convolutional forward pass

Begin by implementing a naive version of the forward pass of the CNN that uses `for` loops. This function is `conv_forward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a triple `for` loop.

After you implement `conv_forward_naive`, test your implementation by running the cell below.

```
In [2]:  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
```

Convolutional backward pass

Now, implement a naive version of the backward pass of the CNN. The function is `conv_backward_naive` in `nndl/conv_layers.py`. Don't worry about efficiency of implementation. Later on, we provide a fast implementation of these layers. This version ought to test your understanding of convolution. In our implementation, there is a quadruple `for` loop.

After you implement `conv_backward_naive`, test your implementation by running the cell below.

```
In [3]:  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}

         out, cache = conv_forward_naive(x,w,b,conv_param)

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

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

```
Testing conv_backward_naive function
dx error:   1.0774825498644653e-09
dw error:   1.136827566794141e-09
db error:   6.960861358722253e-12
```

Max pool forward pass

In this section, we will implement the forward pass of the max pool. The function is
`max_pool_forward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of
implementation.

After you implement `max_pool_forward_naive`, test your implementation by running the cell
below.

```
In [4]: 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
```

Max pool backward pass

In this section, you will implement the backward pass of the max pool. The function is `max_pool_backward_naive` in `nndl/conv_layers.py`. Do not worry about the efficiency of implementation.

After you implement `max_pool_backward_naive`, test your implementation by running the cell below.

```
In [5]: 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, 

        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:  3.27564236064436e-12
```

Fast implementation of the CNN layers

Implementing fast versions of the CNN layers can be difficult. We will provide you with the fast layers implemented by cs231n. They are provided in `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
```

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 cell below.

You should see pretty drastic speedups in the implementation of these layers. On our machine, the forward pass speeds up by 17x and the backward pass speeds up by 840x. Of course, these numbers will vary from machine to machine, as well as on your precise implementation of the naive layers.

```
In [6]: from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
        from time import time

        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: 7.882767s
Fast: 0.018243s
Speedup: 432.096423x
Difference:  7.549127760735231e-11

Testing conv_backward_fast:
Naive: 12.341964s
Fast: 0.011131s
Speedup: 1108.810971x
dx difference:  6.026739466465579e-12
dw difference:  1.1720663565135198e-12
db difference:  0.0
```

In [7]:
```python
from cs231n.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: 0.643086s
fast: 0.004398s
speedup: 146.218789x
difference:  0.0

Testing pool_backward_fast:
Naive: 1.999960s
speedup: 150.873955x
dx difference:  0.0
```

Implementation of cascaded layers

We've provided the following functions in `nndl/conv_layer_utils.py` :

- `conv_relu_forward`
- `conv_relu_backward`
- `conv_relu_pool_forward`
- `conv_relu_pool_backward`

These use the fast implementations of the conv net layers. You can test them below:

```
In [8]: from nndl.conv_layer_utils import conv_relu_pool_forward, conv_relu_pool_bac

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

        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:  4.988323206304756e-08
dw error:  8.898502387299626e-10
db error:  1.9539529457907348e-11
```

```
In [9]: from nndl.conv_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,
        dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,
        db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,

        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:  4.05666500391977e-09
dw error:  1.1183421468180746e-09
db error:  8.77072721749371e-12
```

What next?

We saw how helpful batch normalization was for training FC nets. In the next notebook, we'll implement a batch normalization for convolutional neural networks, and then finish off by implementing a CNN to improve our validation accuracy on CIFAR-10.