CNN-Layers

February 25, 2021

0.1 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).

[2]:
```
## 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_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))))
```

0.2 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`.

0.2.1 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.

[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
```

0.2.2 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.

[4]:
```
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,␣
 ↪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 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:  1.0
dw error:  2.8070873679848205e-10
db error:  2.2766771005533667e-11
```

0.2.3 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.

[5]:
```
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
```

0.2.4 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.

[6]:
```
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 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:  3.2756387806169813e-12
```

0.3 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.

[7]:
```python
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: 7.645994s
Fast: 0.014226s
Speedup: 537.476729x
Difference:  2.3164423283468067e-11

Testing conv_backward_fast:
Naive: 10.050777s
Fast: 0.012206s
Speedup: 823.440082x
dx difference:  1.0
dw difference:  1.456656519088377e-11
db difference:  0.0
```

[8]:
```
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: 0.527140s
fast: 0.006076s
speedup: 86.759771x
difference:  0.0
```

```
Testing pool_backward_fast:
Naive: 1.583779s
speedup: 89.626550x
dx difference:  0.0
```

0.4 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:

[12]:
```
from nndl.conv_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_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)

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:  1.1765335829016865e-08
dw error:  9.132958239361986e-10
db error:  7.33615166792925e-12
```

[13]:
```
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,␣
 ↪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)

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:  1.1425877178093623e-09
dw error:  1.9940987496613484e-10
db error:  1.991382404345138e-11
```

0.5 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.

1 conv_layers.py

[ ]:
```
import numpy as np
from nndl.layers import *
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

def conv_forward_naive(x, w, b, conv_param):
  """
  A naive implementation of the forward pass for a convolutional layer.
```

```
The input consists of N data points, each with C channels, height H and width
W. We convolve each input with F different filters, where each filter spans
all C channels and has height HH and width HH.

Input:
- x: Input data of shape (N, C, H, W)
- w: Filter weights of shape (F, C, HH, WW)
- b: Biases, of shape (F,)
- conv_param: A dictionary with the following keys:
  - 'stride': The number of pixels between adjacent receptive fields in the
    horizontal and vertical directions.
  - 'pad': The number of pixels that will be used to zero-pad the input.

Returns a tuple of:
- out: Output data, of shape (N, F, H', W') where H' and W' are given by
  H' = 1 + (H + 2 * pad - HH) / stride
  W' = 1 + (W + 2 * pad - WW) / stride
- cache: (x, w, b, conv_param)
"""
out = None
pad = conv_param['pad']
stride = conv_param['stride']

# ================================================================ #
# YOUR CODE HERE:
#   Implement the forward pass of a convolutional neural network.
#   Store the output as 'out'.
#   Hint: to pad the array, you can use the function np.pad.
# ================================================================ #

x_padded = np.pad(x, ((0,0), (0,0), (pad, pad), (pad, pad)), 'constant')

N, _, H, W = x_padded.shape
F, _, HH, WW = w.shape

H_out = int(1 + (H - HH) / stride)
W_out = int(1 + (W - WW) / stride)

out = np.zeros((N, F, H_out, W_out))

for pt_idx in range(N):
    for filter_idx in range(F):
          for x_idx in range(W_out):
              for y_idx in range(H_out):
                  x_start = x_idx * stride
                  y_start = y_idx * stride
```

```python
                    patch = x_padded[pt_idx, :, y_start:y_start+HH, x_start:
 ↪x_start+WW]
                    convolved = np.sum(np.multiply(patch, w[filter_idx])) +␣
 ↪b[filter_idx]
                    out[pt_idx, filter_idx, y_idx, x_idx] = convolved

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

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


def conv_backward_naive(dout, cache):
  """
  A naive implementation of the backward pass for a convolutional layer.

  Inputs:
  - dout: Upstream derivatives.
  - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

  Returns a tuple of:
  - dx: Gradient with respect to x
  - dw: Gradient with respect to w
  - db: Gradient with respect to b
  """
  dx, dw, db = None, None, None

  N, F, out_height, out_width = dout.shape
  x, w, b, conv_param = cache

  stride, pad = [conv_param['stride'], conv_param['pad']]
  xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
  num_filts, _, f_height, f_width = w.shape

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the backward pass of a convolutional neural network.
  #   Calculate the gradients: dx, dw, and db.
  # ================================================================ #

  dx = np.zeros(xpad.shape)
  dw = np.zeros(w.shape)
  db = np.zeros(b.shape)

  for pt_idx in range(N):
```

```
    for filter_idx in range(F):
          db[filter_idx] += np.sum(dout[pt_idx, filter_idx])
          for x_idx in range(out_width):
              for y_idx in range(out_height):
                  x_start = x_idx * stride
                  y_start = y_idx * stride
                  patch = xpad[pt_idx, :, y_start:y_start+f_height, x_start:
↪x_start+f_width]

                  dx[filter_idx, :, y_start:y_start+f_height, x_start:
↪x_start+f_width] += w[filter_idx] + dout[pt_idx, filter_idx, y_idx, x_idx]
                  dw[filter_idx] += patch * dout[pt_idx, filter_idx, y_idx,␣
↪x_idx]

  dx = dx[:, :, pad:pad+x.shape[2], pad:pad+x.shape[3]]


  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  return dx, dw, db


def max_pool_forward_naive(x, pool_param):
  """
  A naive implementation of the forward pass for a max pooling layer.

  Inputs:
  - x: Input data, of shape (N, C, H, W)
  - pool_param: dictionary with the following keys:
    - 'pool_height': The height of each pooling region
    - 'pool_width': The width of each pooling region
    - 'stride': The distance between adjacent pooling regions

  Returns a tuple of:
  - out: Output data
  - cache: (x, pool_param)
  """
  out = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling forward pass.
  # ================================================================ #

  N, C, H, W = x.shape
```

```
  pool_height, pool_width, stride = [pool_param['pool_height'],␣
↪pool_param['pool_width'], pool_param['stride']]
  H_out = int(1 + (H-pool_param['pool_height']) / pool_param['stride'])
  W_out = int(1 + (W-pool_param['pool_width']) / pool_param['stride'])

  out = np.zeros((N, C, H_out, W_out))

  for pt_idx in range(N):
    for channel_idx in range(C):
        for y_idx in range(H_out):
            for x_idx in range(W_out):
                x_start = x_idx * stride
                y_start = y_idx * stride
                out[pt_idx, channel_idx, y_idx, x_idx] = np.max(x[pt_idx,␣
↪channel_idx, y_start:y_start+pool_height, x_start:x_start+pool_width])

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #
  cache = (x, pool_param)
  return out, cache

def max_pool_backward_naive(dout, cache):
  """
  A naive implementation of the backward pass for a max pooling layer.

  Inputs:
  - dout: Upstream derivatives
  - cache: A tuple of (x, pool_param) as in the forward pass.

  Returns:
  - dx: Gradient with respect to x
  """
  dx = None
  x, pool_param = cache
  pool_height, pool_width, stride = pool_param['pool_height'],␣
↪pool_param['pool_width'], pool_param['stride']

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the max pooling backward pass.
  # ================================================================ #

  N, C, H, W = dout.shape
  pool_height, pool_width, stride = [pool_param['pool_height'],␣
↪pool_param['pool_width'], pool_param['stride']]
  dx = np.zeros(x.shape)
```

```python
    for pt_idx in range(N):
      for channel_idx in range(C):
          for y_idx in range(H):
              for x_idx in range(W):
                  x_start = x_idx * stride
                  y_start = y_idx * stride
                  patch = x[pt_idx, channel_idx, y_start:y_start+pool_height,␣
↪x_start:x_start+pool_width]
                  dx[pt_idx, channel_idx, y_start:y_start+pool_height, x_start:
↪x_start+pool_width] += (patch == np.max(patch)) * dout[pt_idx, channel_idx,␣
↪y_idx, x_idx]

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx

def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """
    Computes the forward pass for spatial batch normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance. momentum=0 means that
        old information is discarded completely at every time step, while
        momentum=1 means that new information is never incorporated. The
        default of momentum=0.9 should work well in most situations.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the spatial batchnorm forward pass.
    #
```

```python
    #   You may find it useful to use the batchnorm forward pass you
    #   implemented in HW #4.
    # ================================================================ #

    N, C, H, W = x.shape
    x = x.transpose(0, 2, 3, 1).reshape((-1, C))
    out, cache = batchnorm_forward(x, gamma, beta, bn_param)
    out = out.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return out, cache


def spatial_batchnorm_backward(dout, cache):
    """
    Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the spatial batchnorm backward pass.
    #
    #   You may find it useful to use the batchnorm forward pass you
    #   implemented in HW #4.
    # ================================================================ #

    N, C, H, W = dout.shape
    dout = dout.transpose(0, 2, 3, 1).reshape((-1, C))
    dx, dgamma, dbeta = batchnorm_backward(dout, cache)
    dx = dx.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

```python
    return dx, dgamma, dbeta
```