from \_\_future\_\_ import print\_function

import numpy as np

try:

from drive.Stanford.CS231n\_master.assignment2.cs231n.im2col\_cython import col2im\_cython, im2col\_cython

from drive.Stanford.CS231n\_master.assignment2.cs231n.im2col\_cython import col2im\_6d\_cython

except ImportError:

print('run the following from the cs231n directory and try again:')

print('python setup.py build\_ext --inplace')

print('You may also need to restart your iPython kernel')

from cs231n.im2col import \*

def conv\_forward\_im2col(x, w, b, conv\_param):

"""

A fast implementation of the forward pass for a convolutional layer

based on im2col and col2im.

"""

N, C, H, W = x.shape

num\_filters, \_, filter\_height, filter\_width = w.shape

stride, pad = conv\_param['stride'], conv\_param['pad']

# Check dimensions

assert (W + 2 \* pad - filter\_width) % stride == 0, 'width does not work'

assert (H + 2 \* pad - filter\_height) % stride == 0, 'height does not work'

# Create output

out\_height = (H + 2 \* pad - filter\_height) // stride + 1

out\_width = (W + 2 \* pad - filter\_width) // stride + 1

out = np.zeros((N, num\_filters, out\_height, out\_width), dtype=x.dtype)

# x\_cols = im2col\_indices(x, w.shape[2], w.shape[3], pad, stride)

x\_cols = im2col\_cython(x, w.shape[2], w.shape[3], pad, stride)

res = w.reshape((w.shape[0], -1)).dot(x\_cols) + b.reshape(-1, 1)

out = res.reshape(w.shape[0], out.shape[2], out.shape[3], x.shape[0])

out = out.transpose(3, 0, 1, 2)

cache = (x, w, b, conv\_param, x\_cols)

return out, cache

def conv\_forward\_strides(x, w, b, conv\_param):

N, C, H, W = x.shape

F, \_, HH, WW = w.shape

stride, pad = conv\_param['stride'], conv\_param['pad']

# Check dimensions

#assert (W + 2 \* pad - WW) % stride == 0, 'width does not work'

#assert (H + 2 \* pad - HH) % stride == 0, 'height does not work'

# Pad the input

p = pad

x\_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

# Figure out output dimensions

H += 2 \* pad

W += 2 \* pad

out\_h = (H - HH) // stride + 1

out\_w = (W - WW) // stride + 1

# Perform an im2col operation by picking clever strides

shape = (C, HH, WW, N, out\_h, out\_w)

strides = (H \* W, W, 1, C \* H \* W, stride \* W, stride)

strides = x.itemsize \* np.array(strides)

x\_stride = np.lib.stride\_tricks.as\_strided(x\_padded,

shape=shape, strides=strides)

x\_cols = np.ascontiguousarray(x\_stride)

x\_cols.shape = (C \* HH \* WW, N \* out\_h \* out\_w)

# Now all our convolutions are a big matrix multiply

res = w.reshape(F, -1).dot(x\_cols) + b.reshape(-1, 1)

# Reshape the output

res.shape = (F, N, out\_h, out\_w)

out = res.transpose(1, 0, 2, 3)

# Be nice and return a contiguous array

# The old version of conv\_forward\_fast doesn't do this, so for a fair

# comparison we won't either

out = np.ascontiguousarray(out)

cache = (x, w, b, conv\_param, x\_cols)

return out, cache

def conv\_backward\_strides(dout, cache):

x, w, b, conv\_param, x\_cols = cache

stride, pad = conv\_param['stride'], conv\_param['pad']

N, C, H, W = x.shape

F, \_, HH, WW = w.shape

\_, \_, out\_h, out\_w = dout.shape

db = np.sum(dout, axis=(0, 2, 3))

dout\_reshaped = dout.transpose(1, 0, 2, 3).reshape(F, -1)

dw = dout\_reshaped.dot(x\_cols.T).reshape(w.shape)

dx\_cols = w.reshape(F, -1).T.dot(dout\_reshaped)

dx\_cols.shape = (C, HH, WW, N, out\_h, out\_w)

dx = col2im\_6d\_cython(dx\_cols, N, C, H, W, HH, WW, pad, stride)

return dx, dw, db

def conv\_backward\_im2col(dout, cache):

"""

A fast implementation of the backward pass for a convolutional layer

based on im2col and col2im.

"""

x, w, b, conv\_param, x\_cols = cache

stride, pad = conv\_param['stride'], conv\_param['pad']

db = np.sum(dout, axis=(0, 2, 3))

num\_filters, \_, filter\_height, filter\_width = w.shape

dout\_reshaped = dout.transpose(1, 2, 3, 0).reshape(num\_filters, -1)

dw = dout\_reshaped.dot(x\_cols.T).reshape(w.shape)

dx\_cols = w.reshape(num\_filters, -1).T.dot(dout\_reshaped)

# dx = col2im\_indices(dx\_cols, x.shape, filter\_height, filter\_width, pad, stride)

dx = col2im\_cython(dx\_cols, x.shape[0], x.shape[1], x.shape[2], x.shape[3],

filter\_height, filter\_width, pad, stride)

return dx, dw, db

conv\_forward\_fast = conv\_forward\_strides

conv\_backward\_fast = conv\_backward\_strides

def max\_pool\_forward\_fast(x, pool\_param):

"""

A fast implementation of the forward pass for a max pooling layer.

This chooses between the reshape method and the im2col method. If the pooling

regions are square and tile the input image, then we can use the reshape

method which is very fast. Otherwise we fall back on the im2col method, which

is not much faster than the naive method.

"""

N, C, H, W = x.shape

pool\_height, pool\_width = pool\_param['pool\_height'], pool\_param['pool\_width']

stride = pool\_param['stride']

same\_size = pool\_height == pool\_width == stride

tiles = H % pool\_height == 0 and W % pool\_width == 0

if same\_size and tiles:

out, reshape\_cache = max\_pool\_forward\_reshape(x, pool\_param)

cache = ('reshape', reshape\_cache)

else:

out, im2col\_cache = max\_pool\_forward\_im2col(x, pool\_param)

cache = ('im2col', im2col\_cache)

return out, cache

def max\_pool\_backward\_fast(dout, cache):

"""

A fast implementation of the backward pass for a max pooling layer.

This switches between the reshape method an the im2col method depending on

which method was used to generate the cache.

"""

method, real\_cache = cache

if method == 'reshape':

return max\_pool\_backward\_reshape(dout, real\_cache)

elif method == 'im2col':

return max\_pool\_backward\_im2col(dout, real\_cache)

else:

raise ValueError('Unrecognized method "%s"' % method)

def max\_pool\_forward\_reshape(x, pool\_param):

"""

A fast implementation of the forward pass for the max pooling layer that uses

some clever reshaping.

This can only be used for square pooling regions that tile the input.

"""

N, C, H, W = x.shape

pool\_height, pool\_width = pool\_param['pool\_height'], pool\_param['pool\_width']

stride = pool\_param['stride']

assert pool\_height == pool\_width == stride, 'Invalid pool params'

assert H % pool\_height == 0

assert W % pool\_height == 0

x\_reshaped = x.reshape(N, C, H // pool\_height, pool\_height,

W // pool\_width, pool\_width)

out = x\_reshaped.max(axis=3).max(axis=4)

cache = (x, x\_reshaped, out)

return out, cache

def max\_pool\_backward\_reshape(dout, cache):

"""

A fast implementation of the backward pass for the max pooling layer that

uses some clever broadcasting and reshaping.

This can only be used if the forward pass was computed using

max\_pool\_forward\_reshape.

NOTE: If there are multiple argmaxes, this method will assign gradient to

ALL argmax elements of the input rather than picking one. In this case the

gradient will actually be incorrect. However this is unlikely to occur in

practice, so it shouldn't matter much. One possible solution is to split the

upstream gradient equally among all argmax elements; this should result in a

valid subgradient. You can make this happen by uncommenting the line below;

however this results in a significant performance penalty (about 40% slower)

and is unlikely to matter in practice so we don't do it.

"""

x, x\_reshaped, out = cache

dx\_reshaped = np.zeros\_like(x\_reshaped)

out\_newaxis = out[:, :, :, np.newaxis, :, np.newaxis]

mask = (x\_reshaped == out\_newaxis)

dout\_newaxis = dout[:, :, :, np.newaxis, :, np.newaxis]

dout\_broadcast, \_ = np.broadcast\_arrays(dout\_newaxis, dx\_reshaped)

dx\_reshaped[mask] = dout\_broadcast[mask]

dx\_reshaped /= np.sum(mask, axis=(3, 5), keepdims=True)

dx = dx\_reshaped.reshape(x.shape)

return dx

def max\_pool\_forward\_im2col(x, pool\_param):

"""

An implementation of the forward pass for max pooling based on im2col.

This isn't much faster than the naive version, so it should be avoided if

possible.

"""

N, C, H, W = x.shape

pool\_height, pool\_width = pool\_param['pool\_height'], pool\_param['pool\_width']

stride = pool\_param['stride']

assert (H - pool\_height) % stride == 0, 'Invalid height'

assert (W - pool\_width) % stride == 0, 'Invalid width'

out\_height = (H - pool\_height) // stride + 1

out\_width = (W - pool\_width) // stride + 1

x\_split = x.reshape(N \* C, 1, H, W)

x\_cols = im2col(x\_split, pool\_height, pool\_width, padding=0, stride=stride)

x\_cols\_argmax = np.argmax(x\_cols, axis=0)

x\_cols\_max = x\_cols[x\_cols\_argmax, np.arange(x\_cols.shape[1])]

out = x\_cols\_max.reshape(out\_height, out\_width, N, C).transpose(2, 3, 0, 1)

cache = (x, x\_cols, x\_cols\_argmax, pool\_param)

return out, cache

def max\_pool\_backward\_im2col(dout, cache):

"""

An implementation of the backward pass for max pooling based on im2col.

This isn't much faster than the naive version, so it should be avoided if

possible.

"""

x, x\_cols, x\_cols\_argmax, pool\_param = cache

N, C, H, W = x.shape

pool\_height, pool\_width = pool\_param['pool\_height'], pool\_param['pool\_width']

stride = pool\_param['stride']

dout\_reshaped = dout.transpose(2, 3, 0, 1).flatten()

dx\_cols = np.zeros\_like(x\_cols)

dx\_cols[x\_cols\_argmax, np.arange(dx\_cols.shape[1])] = dout\_reshaped

dx = col2im\_indices(dx\_cols, (N \* C, 1, H, W), pool\_height, pool\_width,

padding=0, stride=stride)

dx = dx.reshape(x.shape)

return dx