convolutation_matmul

January 17, 2023

1 Convolution and Matrix Multiplication

The convolution is a well known image transformation used to transform an image. It can be used to blur, to compute the gradient in one direction and it is widely used in deep neural networks. Having a fast implementation is important.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
%matplotlib inline
```

[3]:
```
%load_ext mlprodict
```

1.1 numpy

Image have often 4 dimensions (N, C, H, W) = (batch, channels, height, width). Let's first start with a 2D image.

[4]:
```
import numpy

shape = (5, 7)
N = numpy.prod(shape)
data = numpy.arange(N).astype(numpy.float32).reshape(shape)
# data[:, :] = 0
# data[2, 3] = 1
data.shape
```

[4]:
```
(5, 7)
```

Let's a 2D kernel, the same one.

[5]:
```
kernel = (numpy.arange(9) + 1).reshape(3, 3).astype(numpy.float32)
kernel
```

[5]:
```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]], dtype=float32)
```

1.1.1 raw convolution

A raw version of a 2D convolution.

[6]:
```
def raw_convolution(data, kernel):
    rx = (kernel.shape[0] - 1) // 2
    ry = (kernel.shape[1] - 1) // 2
    res = numpy.zeros(data.shape, dtype=data.dtype)
    for i in range(data.shape[0]):
        for j in range(data.shape[1]):
            for x in range(kernel.shape[0]):
                for y in range(kernel.shape[1]):
                    a = i + x - rx
                    b = j + y - ry
                    if a < 0 or b < 0 or a >= data.shape[0] or b >= data.shape[1]:
                        continue
                    res[i, j] += kernel[x, y] * data[a, b]
    return res

res = raw_convolution(data, kernel)
res.shape
```

[6]:
```
(5, 7)
```

[7]:
```
res
```

[7]:
```
array([[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
       [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
       [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
       [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
       [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]], dtype=float32)
```

1.1.2 With pytorch

pytorch is optimized for deep learning and prefers 4D tenors to represent multiple images. We add two empty dimension to the previous example.

[8]:
```
from torch import from_numpy
from torch.nn.functional import conv2d
```

[9]:
```
rest = conv2d(from_numpy(data[numpy.newaxis, numpy.newaxis, ...]),
              from_numpy(kernel[numpy.newaxis, numpy.newaxis, ...]),
              padding=(1, 1))
rest.shape
```

[9]:
```
torch.Size([1, 1, 5, 7])
```

[10]:
```
rest
```

[10]:
```
tensor([[[[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
          [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
          [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
          [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
          [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]]]])
```

Everything works.

[11]:
```
from numpy.testing import assert_almost_equal
assert_almost_equal(res, rest[0, 0].numpy())
```

1.1.3 using Gemm?

A fast implementation could reuse whatever exists with a fast implementation such as a matrix multiplication. The goal is to transform the tensor `data` into a new matrix which can be mutiplied with a flatten kernel and finally reshaped into the expected result. pytorch calls this function Unfold. This function is also called im2col.

[12]:
```
from torch.nn import Unfold
unfold = Unfold(kernel_size=(3, 3), padding=(1, 1))(from_numpy(data[numpy.newaxis,␣
↪numpy.newaxis, ...]))
unfold.shape
```

[12]:
```
torch.Size([1, 9, 35])
```

We then multiply this matrix with the flattened kernel and reshape it.

[13]:
```
impl = kernel.flatten() @ unfold.numpy()
impl = impl.reshape(data.shape)
impl.shape
```

[13]:
```
(5, 7)
```

[14]:
```
impl
```

[14]:
```
array([[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
       [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
       [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
       [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
       [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]], dtype=float32)
```

Everything works as expected.

[15]:
```
assert_almost_equal(res, impl)
```

1.2 What is ConvTranspose?

Deep neural network are trained with a stochastic gradient descent. The gradient of every layer needs to be computed including the gradient of a convolution transpose. That seems easier with the second expression of a convolution relying on a matrix multiplication and function `im2col`. `im2col` is just a new matrix built from `data` where every value was copied in 9=3x3 locations. The gradient against an input value `data[i,j]` is the sum of 9=3x3 values from the output gradient. If `im2col` plays with indices, the gradient requires to do the same thing in the other way.

[16]:
```
# impl[:, :] = 0
# impl[2, 3] = 1
impl
```

[16]:
```
array([[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
       [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
       [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
       [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
       [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]], dtype=float32)
```

[17]:
```
from torch.nn.functional import conv_transpose2d

ct = conv_transpose2d(from_numpy(impl.reshape(data.shape)[numpy.newaxis, numpy.
 ↪newaxis, ...]),
                      from_numpy(kernel[numpy.newaxis, numpy.newaxis, ...]),
                      padding=(1, 1)).numpy()
ct
```

[17]:
```
array([[[[ 2672.,  5379.,  6804.,  7659.,  8514.,  8403.,  6254.],
         [ 8117., 15408., 18909., 20790., 22671., 21780., 15539.],
         [14868., 27315., 32400., 34425., 36450., 34191., 23922.],
         [20039., 35544., 41283., 43164., 45045., 41508., 28325.],
         [18608., 32055., 36756., 38151., 39546., 35943., 23966.]]]],
      dtype=float32)
```

And now the version with `col2im` or Fold applied on the result product of the output from `Conv` and the kernel: the output of `Conv` is multiplied by every coefficient of the kernel. Then all these matrices are concatenated to build a matrix of the same shape of `unfold`.

[18]:
```
p = kernel.flatten().reshape((-1, 1)) @ impl.flatten().reshape((1, -1))
p.shape
```

[18]:
```
(9, 35)
```

[19]:
```
from torch.nn import Fold

fold = Fold(kernel_size=(3, 3), output_size=(5, 7), padding=(1, 1))(from_numpy(p[numpy.
 ↪newaxis, ...]))
fold.shape
```

[19]:
```
torch.Size([1, 1, 5, 7])
```

[20]:
```
fold
```

[20]:
```
tensor([[[[ 2672.,  5379.,  6804.,  7659.,  8514.,  8403.,  6254.],
          [ 8117., 15408., 18909., 20790., 22671., 21780., 15539.],
          [14868., 27315., 32400., 34425., 36450., 34191., 23922.],
          [20039., 35544., 41283., 43164., 45045., 41508., 28325.],
          [18608., 32055., 36756., 38151., 39546., 35943., 23966.]]]])
```

1.3 onnxruntime-training

Following lines shows how *onnxruntime* handles the gradient computation. This section still needs work.

1.3.1 Conv

[21]:
```
from mlprodict.npy.xop import loadop
OnnxConv = loadop(('', 'Conv'))
node = OnnxConv('X', kernel[numpy.newaxis, numpy.newaxis, ...], pads=[1, 1, 1, 1])
onx = node.to_onnx(numpy.float32, numpy.float32)
%onnxview onx
```

```
No CUDA runtime is found, using CUDA_HOME='C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v11.5'
```

[21]: <jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x1ab4521ae90>

[22]:
```
from mlprodict.onnxrt import OnnxInference
oinf = OnnxInference(onx, runtime='onnxruntime1')
oinf.run({'X': data[numpy.newaxis, numpy.newaxis, ...]})['out_con_0']
```

[22]:
```
array([[[[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
         [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
         [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
         [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
         [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]]]],
      dtype=float32)
```

It is the same.

[23]:
```
from onnxcustom.training.grad_helper import onnx_derivative, DerivativeOptions
grad = onnx_derivative(onx, options=DerivativeOptions.FillGrad | DerivativeOptions.
 ↪KeepOutputs)
```

[24]:
```
%onnxview grad
```

[24]: <jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x1ab45f003d0>

[25]:
```
oinf = OnnxInference(grad, runtime='onnxruntime1')
```

[26]:
```
res = oinf.run({'X': data[numpy.newaxis, numpy.newaxis, ...],
                'init': kernel[numpy.newaxis, numpy.newaxis, ...]})
res
```

[26]:
```
{'X_grad': array([[[[12., 21., 21., 21., 21., 21., 16.],
          [27., 45., 45., 45., 45., 45., 33.],
          [27., 45., 45., 45., 45., 45., 33.],
          [27., 45., 45., 45., 45., 45., 33.],
          [24., 39., 39., 39., 39., 39., 28.]]]], dtype=float32),
 'init_grad': array([[[[312., 378., 336.],
          [495., 595., 525.],
          [480., 574., 504.]]]], dtype=float32),
 'out_con_0': array([[[[ 134.,  211.,  250.,  289.,  328.,  367.,  238.],
          [ 333.,  492.,  537.,  582.,  627.,  672.,  423.],
          [ 564.,  807.,  852.,  897.,  942.,  987.,  612.],
          [ 795., 1122., 1167., 1212., 1257., 1302.,  801.],
          [ 422.,  571.,  592.,  613.,  634.,  655.,  382.]]]],
       dtype=float32)}
```

1.3.2 ConvTranspose

[27]:
```
from mlprodict.npy.xop import loadop

OnnxConvTranspose = loadop('ConvTranspose')
node = OnnxConvTranspose('X', kernel[numpy.newaxis, numpy.newaxis, ...], pads=[1, 1,␣
 ↪1, 1])
onx = node.to_onnx(numpy.float32, numpy.float32)
%onnxview onx
```

[27]: <jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x1ab45f00fd0>

[28]:
```
oinf = OnnxInference(onx, runtime='onnxruntime1')
ct = oinf.run({'X': impl[numpy.newaxis, numpy.newaxis, ...]})['out_con_0']
ct
```

[28]:
```
array([[[[ 2672.,  5379.,  6804.,  7659.,  8514.,  8403.,  6254.],
         [ 8117., 15408., 18909., 20790., 22671., 21780., 15539.],
         [14868., 27315., 32400., 34425., 36450., 34191., 23922.],
         [20039., 35544., 41283., 43164., 45045., 41508., 28325.],
         [18608., 32055., 36756., 38151., 39546., 35943., 23966.]]]],
      dtype=float32)
```

1.4 im2col and col2im

Function `im2col` transforms an image so that the convolution of this image can be expressed as a matrix multiplication. It takes the image and the kernel shape.

[29]:
```
from mlprodict.onnxrt.ops_cpu.op_conv_helper import im2col

v = numpy.arange(5).astype(numpy.float32)
w = im2col(v, (3, ))
w
```

[29]:
```
array([[0., 0., 1.],
       [0., 1., 2.],
       [1., 2., 3.],
       [2., 3., 4.],
       [3., 4., 0.]], dtype=float32)
```

[30]:
```
k = numpy.array([1, 1, 1], dtype=numpy.float32)
conv = w @ k
conv
```

[30]:
```
array([1., 3., 6., 9., 7.], dtype=float32)
```

Let's compare with the numpy function.

[31]:
```
numpy.convolve(v, k, mode='same')
```

[31]:
```
array([1., 3., 6., 9., 7.], dtype=float32)
```

$conv(v,k) = im2col(v, shape(k))\ k = w\ k$ where $w = im2col(v, shape(k))$.
In deep neural network, the gradient is propagated from the last layer to the first one. At some point, the backpropagation produces the gradient $\frac{d(E)}{d(conv)}$, the gradient of the error against the outputs of the convolution layer. Then $\frac{d(E)}{d(v)} = \frac{d(E)}{d(conv(v,k))} \frac{d(conv(v,k))}{d(v)}$.
We need to compute $\frac{d(conv(v,k))}{d(v)} = \frac{d(conv(v,k))}{d(w)} \frac{d(w)}{d(v)}$.
We can say that $\frac{d(conv(v,k))}{d(w)} = k$.
That leaves $\frac{d(w)}{d(v)} = \frac{d(im2col(v,shape(k)))}{d(v)}$. And this last term is equal to $im2col(m, shape(k))$ where m is a matrix identical to v except that all not null parameter are replaced by 1. To summarize: $\frac{d(im2col(v,shape(k)))}{d(v)} = im2col(v \neq 0, shape(k))$.
Finally, $\frac{d(E)}{d(v)} = \frac{d(E)}{d(conv(v,k))} \frac{d(conv(v,k))}{d(v)} = \frac{d(E)}{d(conv(v,k))}\ k\ im2col(v \neq 0, shape(k))$.

Now, $im2col(v \neq 0, shape(k))$ is a very simple matrix with only ones or zeros. Is there a way we can avoid doing the matrix multiplication but simply adding terms? That's the purpose of function $col2im$ defined so that:
$\frac{d(E)}{d(v)} = \frac{d(E)}{d(conv(v,k))}\ k\ im2col(v \neq 0, shape(k)) = col2im\left(\frac{d(E)}{d(conv(v,k))}\ k, shape(k)\right)$

[32]:

```
```