Convolution and Transposed Convolution as Matrix Multiplication

Introduction

In deep learning, convolution and transposed convolution are often used in the neural networks. Unlike convolution, transposed convolution is sometimes confusing and not too many people know why it's called transposed convolution.

In this blog post, I would like to discuss how to view convolution and transposed convolution as matrix multiplication, and how to understand the name of transposed convolution.

Convolution and Transposed Convolution as Matrix Multiplication

At each step of the convolution operation, we apply the kernel tensor onto a portion of the elements in the input tensor, compute their element-wise products, and further sum the products together. We repeat the step many times until the kernel convolves over the entire tensor. With this in mind, we could actually formulate the convolution operation using matrix operations equivalently.

Concretely, for any convolution operation in deep learning, $Y = K * X$, it could be formulated as matrix multiplication

where $X^{'}$ is the flatten representation of X which might have already been padded and dilated, W is the sparse matrix representation of kernel K, and $Y^{'}$ is the flatten representation of the output Y.

More specifically, when K and X are 2D matrices, $K \in R^{h_K \times w_K}$, $X \in R^{h_X \times w_X}$, $X^{'} \in R^{h_X w_X}$, $Y \in R^{h_Y \times w_Y}$, $Y \in R^{h_Y w_Y}$, we should have a sparse matrix $W \in R^{h_Y w_Y \times h_X w_X}$ that does the same transformation as the convolution.

Similarly, for any transposed convolution operation in deep learning, $Z = K \star Y$, it could also be formulated as matrix multiplication

where $Z^{'}$ is the flatten representation of the output Z, Y and $Y^{'}$ are just ones we just obtained from the previous convolution operation $Y = K * X$.

Z must have $Z \in R^{h_X \times w_X}$ and $Z^{'} \in R^{h_X w_X}$, as if the shape have been reverted back before the convolution operation $Y = K * X$. Notice that the kernels used for the convolution and transposed convolution, K, are exactly the same, and the weight matrices used in the convolution and transposed convolution matrix multiplications, W and $W^{\top}$, are just transposed to each other. Z, however, usually does not equal to X. Because $Z \neq X$, we cannot call transposed convolution as deconvolution.

Implementing Convolution and Transposed Convolution as Matrix Operation

Let's ignore the channel dimension and the bias term for convolution and transposed convolution for now, and implement convolution and transposed convolution as matrix operation. We also assume the stride is 1 for both convolution and transposed convolution.

conv_as_gemm.py

```
import torch
from torch import nn


def corr2d(X, K):

    # Convolution in deep learning is a misnomer.
    # In fact, it is cross-correlation.
    # https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html
    # This is equivalent as Conv2D that that input_channel == output_channel == 1 and stride == 1.

    assert X.dim() == 2 and K.dim() == 2

    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()

    return Y


def get_sparse_kernel_matrix(K, h_X, w_X):

    # Assuming no channels and stride == 1.
    # Convert the kernel matrix to sparse matrix (dense matrix with lots of zeros in fact).
    # This is a little bit brain-twisting.

    h_K, w_K = K.shape

    h_Y, w_Y = h_X - h_K + 1, w_X - w_K + 1

    W = torch.zeros((h_Y * w_Y, h_X * w_X))
    for i in range(h_Y):
        for j in range(w_Y):
            for ii in range(h_K):
                for jj in range(w_K):
                    W[i * w_Y + j, i * w_X + j + ii * w_X + jj] = K[ii, jj]

    return W


def conv2d_as_matrix_mul(X, K):

    # Assuming no channels and stride == 1.
    # Convert the kernel matrix to sparse matrix (dense matrix with lots of zeros in fact).
    # This is a little bit brain-twisting.

    h_K, w_K = K.shape
```

```
    h_X, w_X = X.shape

    h_Y, w_Y = h_X - h_K + 1, w_X - w_K + 1

    W = get_sparse_kernel_matrix(K=K, h_X=h_X, w_X=w_X)

    Y = torch.matmul(W, X.reshape(-1)).reshape(h_Y, w_Y)

    return Y


def conv_transposed_2d_as_matrix_mul(X, K):

    # Assuming no channels and stride == 1.
    # Convert the kernel matrix to sparse matrix (dense matrix with lots of zeros in fact).
    # This is a little bit brain-twisting.

    h_K, w_K = K.shape
    h_X, w_X = X.shape

    h_Y, w_Y = h_X + h_K - 1, w_X + w_K - 1

    # It's like the kernel were applied on the output tensor.
    W = get_sparse_kernel_matrix(K=K, h_X=h_Y, w_X=w_Y)

    # Weight matrix tranposed.
    Y = torch.matmul(W.T, X.reshape(-1)).reshape(h_Y, w_Y)

    return Y


def main():

    X = torch.arange(30).reshape(5, 6).float()
    K = torch.arange(8).reshape(2, 4).float()
    print("X:")
    print(X)
    print("K:")
    print(K)
    print("Cross-Correlation:")
    Y = corr2d(X=X, K=K)
    print(Y)

    conv = nn.Conv2d(in_channels=1,
                     out_channels=1,
                     kernel_size=K.shape,
                     padding=0,
                     stride=1,
                     bias=False)
    conv.weight.data = K.unsqueeze(0).unsqueeze(0)
```

```
    Z1 = conv(X.unsqueeze(0).unsqueeze(0)).squeeze(0).squeeze(0).detach()
    print("Convolution:")
    print(Z1)
    assert torch.equal(Y, Z1)

    print("Convolution as Matrix Multiplication:")
    Z2 = conv2d_as_matrix_mul(X=X, K=K)
    print(Z2)
    assert torch.equal(Y, Z2)

    conv_transposed = nn.ConvTranspose2d(in_channels=1,
                                         out_channels=1,
                                         kernel_size=K.shape,
                                         padding=0,
                                         stride=1,
                                         bias=False)
    conv_transposed.weight.data = K.unsqueeze(0).unsqueeze(0)
    Z3 = conv_transposed(Y.unsqueeze(0).unsqueeze(0)).squeeze(0).squeeze(0).detach()
    print("Transposed Convolution:")
    print(Z3)
    # The shape will "go back".
    assert Z3.shape == X.shape

    print("Transposed Convolution as Matrix Multiplication:")
    Z4 = conv_transposed_2d_as_matrix_mul(X=Y, K=K)
    print(Z4)
    assert torch.equal(Z3, Z4)
    assert Z4.shape == X.shape

    return


if __name__ == "__main__":

    main()
```

We could see that outputs from the matrix multiplication implementations match the expectation.

```
$ python conv_as_gemm.py
X:
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
        [ 6.,  7.,  8.,  9., 10., 11.],
        [12., 13., 14., 15., 16., 17.],
        [18., 19., 20., 21., 22., 23.],
        [24., 25., 26., 27., 28., 29.]])
K:
tensor([[0., 1., 2., 3.],
        [4., 5., 6., 7.]])
Cross-Correlation:
```

```
tensor([[184., 212., 240.],
        [352., 380., 408.],
        [520., 548., 576.],
        [688., 716., 744.]])
Convolution:
tensor([[184., 212., 240.],
        [352., 380., 408.],
        [520., 548., 576.],
        [688., 716., 744.]])
Convolution as Matrix Multiplication:
tensor([[184., 212., 240.],
        [352., 380., 408.],
        [520., 548., 576.],
        [688., 716., 744.]])
Transposed Convolution:
tensor([[   0.,   184.,   580.,  1216.,  1116.,   720.],
        [  736.,  2120.,  4208.,  5984.,  4880.,  2904.],
        [ 1408.,  3800.,  7232., 10016.,  7904.,  4584.],
        [ 2080.,  5480., 10256., 14048., 10928.,  6264.],
        [ 2752.,  6304., 10684., 12832.,  9476.,  5208.]])
Transposed Convolution as Matrix Multiplication:
tensor([[   0.,   184.,   580.,  1216.,  1116.,   720.],
        [  736.,  2120.,  4208.,  5984.,  4880.,  2904.],
        [ 1408.,  3800.,  7232., 10016.,  7904.,  4584.],
        [ 2080.,  5480., 10256., 14048., 10928.,  6264.],
        [ 2752.,  6304., 10684., 12832.,  9476.,  5208.]])
```

For multi-channel kernel, input tensor, and output tensor, the derivation of the weight matrix for matrix multiplication is more complicated, but it does not change the nature that the convolution and transposed convolution can be viewed as matrix multiplications.

Backward Propagation

Considering the convolution as matrix multiplications.

Given the flattened input vector $x \in R^{m_1}$, weight matrix $W \in R^{m_2 \times m_1}$, and the flattened output vector $y \in R^{m_2}$, the matrix multiplication is

Because the gradients matrix with respect to the input x is

where $\nabla_x y \in R^{m_1 \times m_2}$.

In the backward propagation, the input is $\frac{\partial L}{\partial y} \in R^{m_2}$, and the output of the backward propagation is

$$\begin{aligned} \frac{\partial L}{\partial x} &= \nabla_x y \frac{\partial L}{\partial y} \\ &= W^\top \frac{\partial L}{\partial y} \end{aligned}$$

Therefore, in the forward propagation and the backward propagation of the convolution operation, the weight matrices are W and $W^\top$, respectively.

Similarly, considering the transposed convolution as matrix multiplications. In the forward propagation and the backward propagation of the transposed convolution operation, the weight matrices are $W^{\top}$ and W, respectively.

What does it imply? If the convolution and transposed convolution were implemented as matrix multiplication. To compute the gradient matrix W_1 for transposed convolution operation $Z = K \star Y$, if we happen to have already computed the gradient matrix W_2 for $X = K * Z$, we must have $W_1 = W_2^{\top}$. However, I think in practice this property is less useful because usually the kernel used for different layers are different.

Because the weight matrices used in the forward propagation and the backward propagation of a transposed convolution are just the transpose of the weight matrices used in the forward propagation and the backward propagation of a convolution which has the same kernel parameters, that's probably why transposed convolution is called transposed convolution.

Miscellaneous

We could see that without non-linear activation function, a sequence of convolution operations is just a linear function. Therefore, having non-linear activation is also important for convolutional neural networks.

References