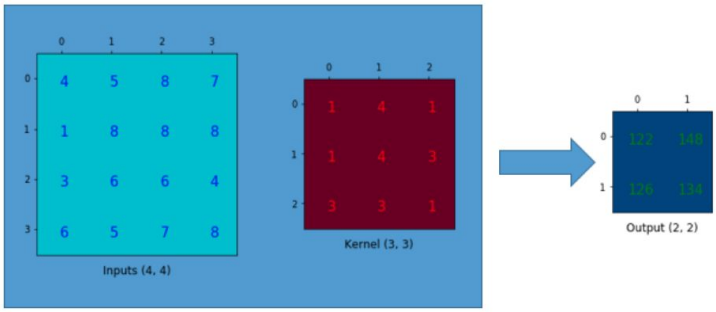
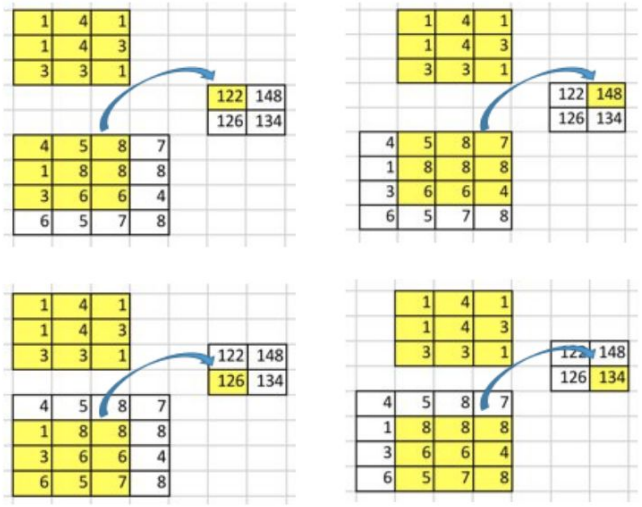
Upsampling with Transposed Convolution

- Transposed convolutions are useful because...
    - The generator in a DCGAN needs to take a few randomly sampled values and upsample it to an image

Standard convolution:

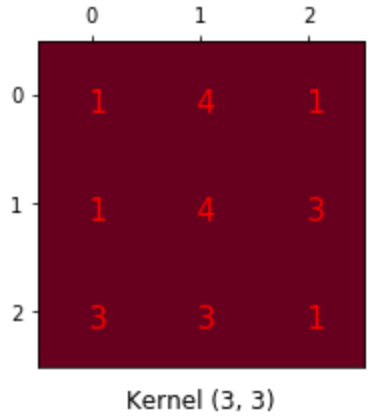




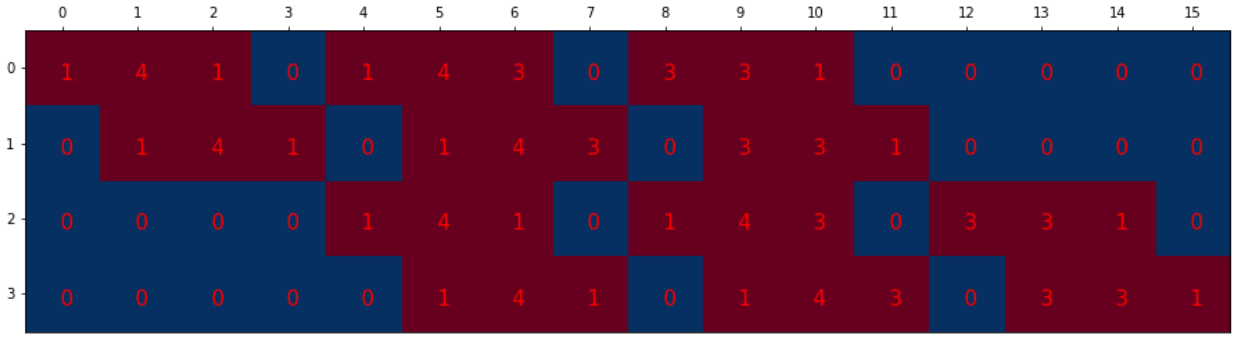
- With convolutions, positional connectivity stays in the output
    - In general, the top left values in input affect top left values in output
- Convolution does a many to one operation (since it does matmul and then sums)

To upsample, we must do this process backwards

Convolution Matrix: (rearrangement of kernel)

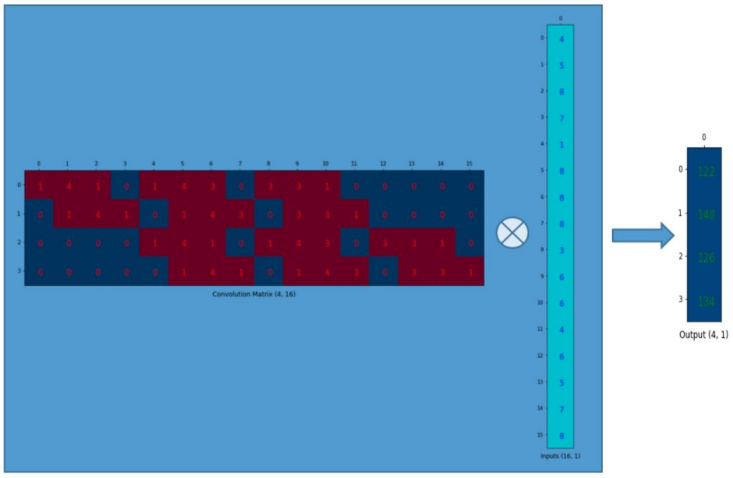


Kernel (3, 3)



Convolution Matrix (4, 16)

- Each row is one convolution operation
    - Each row is the kernel but with 0 padding in different spots
- To use this, first flatten the 4x4 input matrix into a vertical 16 x 1 column vector
- Then multiply the convolution matrix with the column vector, then you get a 4x1 matrix
    - This can be reshaped to a 2x2

- 

- With this convolution matrix, you can go from your 4x4 (=16) matrix to 2x2 (=4) matrix, because the convolution matrix is 4x16.
    - So, if you have a 16x4 matrix, you can go in the reverse direction, from a 2x2 to a 4x4
    - BUT HOW???,,, you use a .....

Transposed Convolutional Matrix:

- We want to go from 2x2 to 4x4, so we used a 16x4 matrix

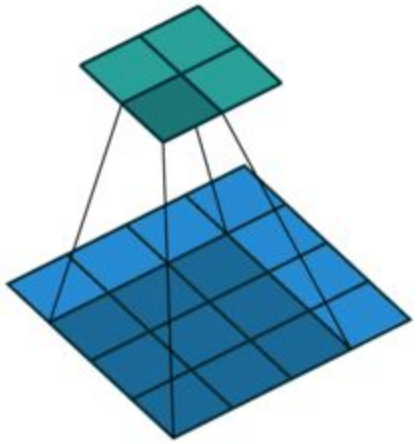
- Just transpose the original 4x16 matrix into 16x4 one.
    - Multiply this by your 4x1 column vector (which is from flattening the 2x2 input)
    - Result is a 16x1 which can be reshaped to a 4x4

Additional Notes

- Weights in the transposed convolution are learnable
- Even though it is called a transposed convolution, it's not like we are taking a matrix and then transposing it, rather we are just doing the opposite of a normal convolution

EXAMPLE:
A = filter
B = input matrix
C = feature map

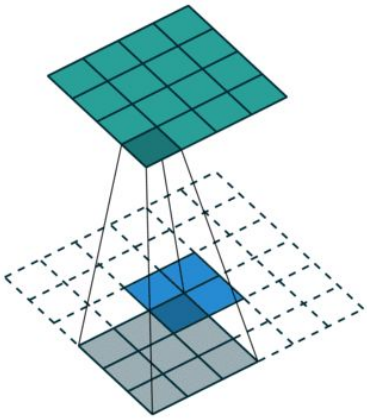
$C = A \times B$
$A^TC = A^T \times A \times B$
$B = A^TC$



Regular convolution:
3x3 kernel on a 4x4 input produces a 2x2 (stride = 1, no padding), since

4 - 3 + 1 = 2

Went from a 4x4 to a 2x2, which means the transpose of this convolution will go from 2x2 to 4x4



You can also do a transposed convolution, using a regular convolution, but less efficient
Padd the input with 2 pixels, so output shape is...
2 + 2(2) - 3 + 1 = 4

Padding must be equal to size of kernel - 1, so that in the case where the kernel is on the top-left of the input, the convolution result only affects the top left pixel of the output
If it were more, then the filter might only see 0s, and if it were less, then you wouldn't have the positional connectivity

Has a lot of useless 0 multiplications... inefficient

Transposed convolution (stride = 1, no padding)

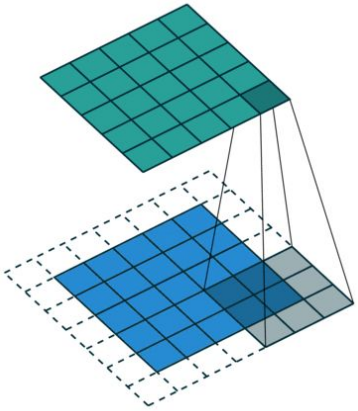
$o_{shape} = i_{shape} + (k - 1)$

- Since the transpose of a non-padded convolution is the same as convolving about a padding input,
    - Then the transpose of a padded convolution is the same as convolving about an input with *less* padding

- **Transposed convolution (stride = 1)**
    - $o_{shape} = i_{shape} + 2p + (k - 1)$

Transposed with SAME (half) padding:

- Since the output size of a half padding convolution is the same as its input size, we can assume that the equivalent of a transpose convolution on a half padded input is the same as a convolution on a half padded input



Input size = 5, kernel = 3, and therefore, padding = 1

K + 2p = i

Transposed with FULL padding:

- Transpose of a fully padded convolution is equivalent to convolving about a non-padded input

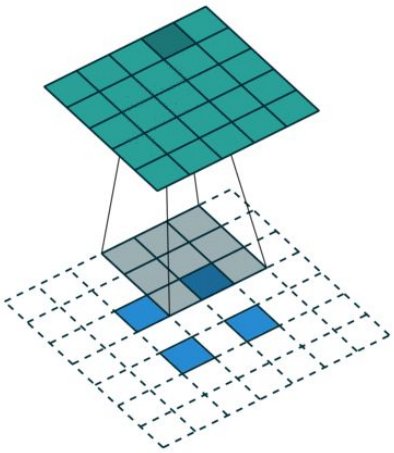
$$o' = i' + (k - 1) - 2p$$

- $$= i' - (k - 1)$$

Transposed, No zero padding, stride > 1:

Transposed of a convolution with stride > 1, is equivalent to a convolution where the stride is less than 1.

- Called a fractionally strided convolution



Zeros are inserted between the input pixels, which in a way, makes the kernel move at a slower stride than 1, because it takes longer to move from one input pixel to another

Math:

If no padding, has a certain stride, and the kernel can fit inside this input, then the transposed convolution can be described as a convolution with....

- s' = 1
- p = k - 1
- i' = adding (s_{org} - 1) zeros between each input pixel

$$\mathbf{o = s(i - 1) + k}$$

Derivation:

Input size ---> add s-1 0s in between each input pixel -----> add kernel size - 1

Adding kernel size only once because you are adding kernel size // 2 to both sides of input img

$$\mathbf{i + (s - 1)(i - 1) + k - 1}$$

$$\mathbf{i + si - s - i + 1 + k - 1}$$

$$\mathbf{si - s + k}$$

$$\mathbf{s(i - 1) + k}$$

Example: above img, has i = 2, k = 3, s = 2

S' = 1

P = 2

Input size = 2

i + (s - 1)(i - 1) + k - 1
2 + (2 - 1)(2 - 1) + 3 - 1
2 + 1 + 3 - 1 === 5 CORRECT

Another example:
Input size = 6x6, no padding, stride = 2, kernel = 4x4

2 (6 - 1) + 4 = 14 CORRECT