# CNN Operators: Convolution, Pooling

Tirtharaj Dash

Dept. of CS & IS and APPCAIR
BITS Pilani, Goa Campus

September 28, 2021

# Where we are

Summary of the previous lecture:

- Adapting fully-connected layers to convolutions
- Convolutions for images

We now know that CNNs are suitable to explore structure in image data, we will understand how various basic CNN operations work in practice:

- Convolution (cross-correlation)
- Padding
- Stride
- pooling

# Convolutions for Images I

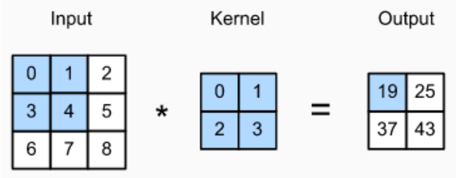
- In our last class we saw how the convolution operation is expressed.
- We also saw that we don't really do "convolution" in CNNs, rather it is simply cross-correlation.
- In general, in a convolutional layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation.

# Convolutions for Images II

- Example:
  - Let us consider a two-dimensional tensor as input with a height of 3 and width of 3 (ignoring change). We mark the shape of the tensor as $3 \times 3$.
  - The height and width of the kernel are both 2.
  - The shape of the kernel window (or convolution window) is given by the height and width of the kernel ($2 \times 2$).

# Convolutions for Images III

- This is how convolution (cross-correlation) is implemented:




- Position the convolution window at the top-left corner of the input tensor and slide it across the input tensor, both from left to right and top to bottom.
- When the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied elementwise and the resulting tensor is summed up yielding a single scalar value.

# Convolutions for Images IV

- This result gives the value of the output tensor at the corresponding location. Here, the output tensor has a height of 2 and width of 2.
- The four elements are derived from the 2D cross-correlation operation are shown below for each $(i, j)$ position in the output matrix:

$$(0,0): 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$
$$(0,1): 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$
$$(1,0): 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$
$$(1,1): 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43$$

# Convolutions for Images V

- Notice that: along each axis, the output size is slightly smaller than the input size.
- Because the kernel has width and height greater than 1, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image.
- The output size is given by the input size $n_h \times n_w$ minus the size of the convolution kernel $k_h \times k_w$:

  $$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

  Example: In our previous example

  $$(3 - 2 + 1) \times (3 - 2 + 1) = 2 \times 2$$

- We can keep the output size same as the input size by *padding* the image with zeros around its boundary so that there is enough space to shift the kernel.

# Convolutional Layers I

- A convolutional layer cross-correlates the input and kernel (parameter) and adds a scalar bias (parameter) to produce an output.
- When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer.
- The kernel update (that is changes to these parameters) will be done based on the standard gradient descent based procedure that we used for MLPs.

# Convolutional Layers II

- First, let's bring our attention to two cases of convolutions where we want to detect horizontal and vertical edges in an image.
- For this, we manually create two edge-detection templates (1 for vertical edges, 1 for horizontal edges), as:

Vertical edge detector:

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

Horizontal edge detector:

$$\begin{bmatrix} +1 & 2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

# Convolutional Layers III

- As we have seen earlier, these detector templates have to be created by a domain-expert. But, there are several questions:
  - How many of these templates are to be created for a problem?
  - How do we know what precisely we are looking for in an image?
  - In a multi-layered convolutional network, the above two questions becomes harder to answer.

# Convolutional Layers IV

- There could be many more such questions that maynot have a definite answer, given that we would be dealing with complex problems in real-world, e.g. detecting a traffic sign in a road imagery.
- To answer these questions, we now have a notion of treating these templates (which we have been calling kernels or filters in CNNs) as *model parameters* and learn them given training data.
- Learning will be again based on gradient-based optimisation procedure (*backpropagation*).

# Learning a Kernel I

- Let X denote the input tensor, and K denote a hand-crafted template, and the output of the convolution (cross-correlation) of X with K is Y (let's call this the target output).
- Let's assume that we are only given $(X, Y)$, and our goal is to learn K.
- How would we do it?

# Learning a Kernel II

- Simple:
  - Randomly initialise a kernel W
  - Use a loss function $L$ between Y and computed output $\hat{\mathrm{Y}}$ (Let $L = (\mathrm{Y} - \hat{\mathrm{Y}})^2$
  - Initialise the learning rate $\alpha$ to some sensible value
  - Repeat:
    - Compute $\hat{\mathrm{Y}} = \texttt{conv2d}(X, W)$
    - Compute $L$
    - $\mathrm{W} \leftarrow \mathrm{W} - \alpha \frac{\partial L}{\partial \mathrm{W}}$

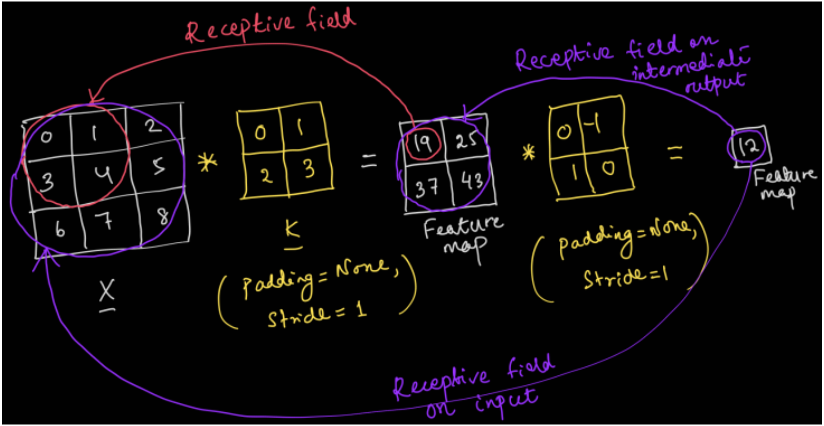
Note:

- Assuming `conv2d` function is implemented with appropriate padding and stride, etc.
- $(\mathrm{Y} - \hat{\mathrm{Y}})^2 = (\mathrm{Y} - \hat{\mathrm{Y}})^{\mathsf{T}}(\mathrm{Y} - \hat{\mathrm{Y}})$, since Y and $\hat{\mathrm{Y}}$ are tensors of same size.

# Learning a Kernel III

- In many cases, just one layer of convolution will not be enough. Most realworld computer vision problems require CNNs to be deeper. In this case, the intermediate convolutional layers maynot have a direct notion of a loss function.
- In such cases, the backpropagation procedure propagates the loss gradients from the outputs towards the inputs via the intermediate layers.
- When any element in a feature map needs a larger receptive field to detect broader features on the input, a deeper network can be considered (see the next slide).

- Receptive field and feature map:



# Padding and Stride I

- So far we have seen convolution operations with no padding and a stride of 1.
- The output size in these cases are: $(n_h - k_h + 1) \times (n_w - k_w + 1)$.
- In several cases, we incorporate techniques, including padding and strided convolutions, that affect the size of the output.

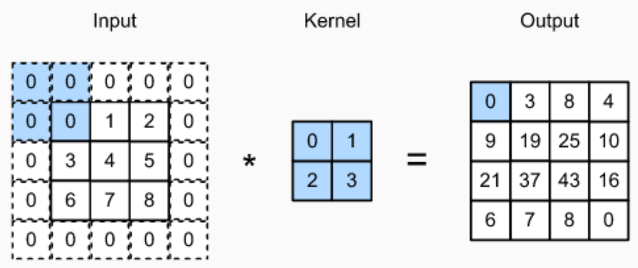
# Padding and Stride II

- Further, in many cases, the information at the boundary of images may be very crucial and this importance is not taken care when doing standard convolution operation.
- To handle information at the boundary of the input and increase the size of the outputs, we apply a method of *padding*, that is: pad the input with 0s in this boundaries.
- In other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be too large. *Strided* convolutions (with stride $> 1$) are a popular technique that can help in these instances.

# Padding and Stride III

Padding:

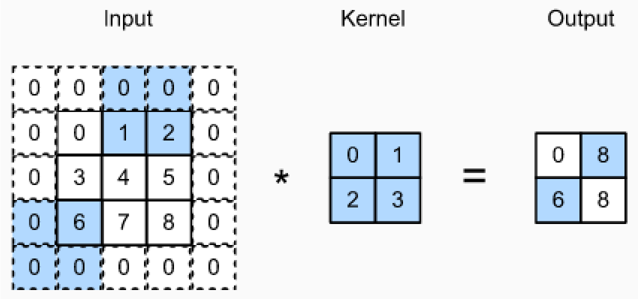
- Let us look at our earlier example with 1 layer of zero padding on input (that is 1 layer on all sides):




- Here the output size is: $(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$ where $(p_h, p_w)$ refers to the padding size on height and width axes respectively.
- Here $(p_h, p_w) = (2, 2)$.

Stride:

- Let us repeat the example in the previous slide with a different stride ($s_h = 3$ along height and $s_w = 2$ along width):



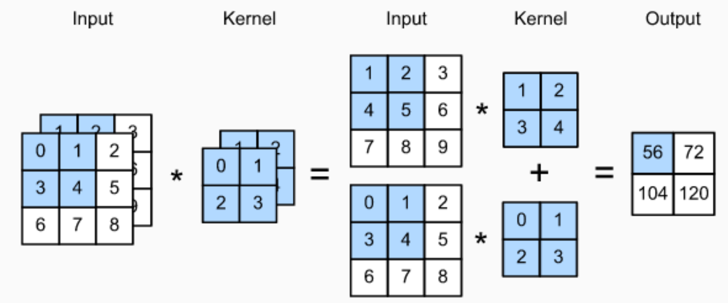

- Output dimension is now:

$$\left\lfloor \frac{(n_h - k_h + p_h + s_h)}{s_h} \times \frac{(n_w - k_w + p_w + s_w)}{s_w} \right\rfloor$$

# Multiple Channels I

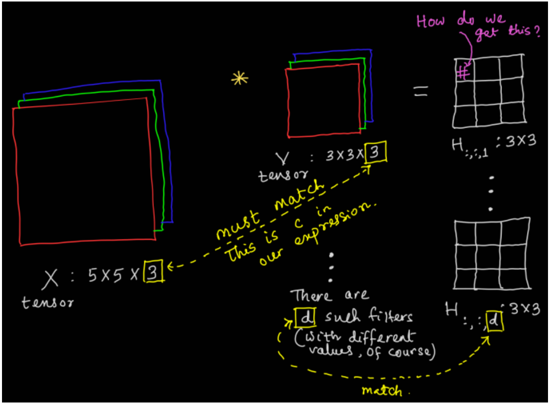
We have already seen these in our last class.

- Multi-channel input:



# Multiple Channels II

- Multi-channel output: H is a volume of feature maps.




(Reused the picture from the last lecture)

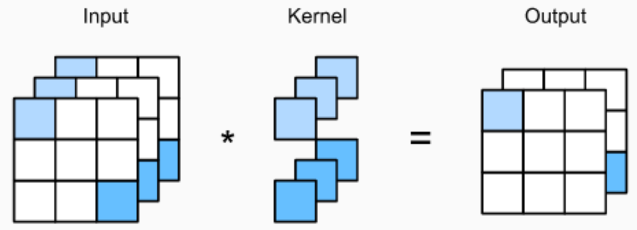
  - Multiple channels can be used to extend the model parameters of the convolutional layer.

# $1 \times 1$ Convolutional Layer I

- We know that convolution correlates adjacent pixels.
- With our present understanding, a $1 \times 1$ kernel does not make much sense.
- However, there are popular operations that are sometimes included in the designs of complex deep CNNs.
- For instance, look at Google's Inception model.

# $1 \times 1$ Convolutional Layer II

- Since the channel dimension is $1 \times 1$ the convolution operations (more correctly, correlation) operations do not have to do anything with the interactions among adjacent pixel elements.
- That is, all that convolution does is on the channel dimension.
- Let's see an example of $1 \times 1$ convolution:



# $1 \times 1$ Convolutional Layer III

- Notice that the input and the output have same height and width.
- $1 \times 1$ convolutional layer can be considered as a fully-connected layer, with weights tied across all the pixel (or indices) locations in the input.
- Practical usage: Often used to reduce the number of depth channels, since it is often very slow to multiply volumes (a bunch of feature maps) with extremely large depths.

# Pooling I

- The hidden representations at the higher layers of a CNN should have lower spatial resolution.
- That is: the each hidden node is sensitive to a larger receptive field in the input.
- We will then be able to ask global questions, such as: "Does this image contain a cat?"
  - Typically the units of our final layer should be sensitive to the entire input.
  - This can be possible by constructing coarser hidden representations.

# Pooling II

- Further, while detecting lower-level features, such as edges, we often want our representations to be translation invariant.
  - If an image is shifted just by 1 pixel, the hidden representations should not change (translation invariant).
  - That is: the hidden representation should not be very sensitive to jitters in the input.

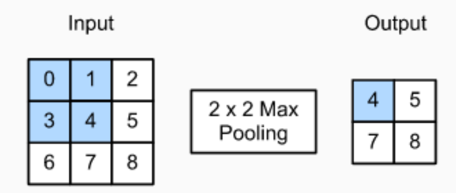
# Pooling III

- Pooling layer:
  - alleviate the excessive sensitivity of the convolutional layer to location
  - spatially downsampling representations
- There are following pooling methods:
  - Max pooling
  - Average pooling
  - $L_p$-norm pooling

  Max and average pooling operations are most frequently used in practice.

# Pooling IV

- Let's look at our example, how this is implemented:




- Notice that, here also there is a concept of padding and stride, used for almost the same reasons as discussed during the convolutions.
- Extension to multiple channel is straightforward: pooling operation works on individual channel; the number of channels before and after pooling is the same.