Computer Vision: 4th lesson – The Sliding Window

In the previous two lessons, we learned about three kinds of operation that carry out feature extraction from an image:

* Filter with a ***convolution*** layer
* Detect with ***ReLU*** activation
* Condense with a ***maximum pooling*** layer

The convolution and pooling operations share a common feature, which both performed over a ***sliding window***. With convolution, this “window” is given by the dimensions of the kernel, the parameter of kernel\_size. With pooling, it is the pooling window, given by pool\_size.

There are two additional parameters affecting both convolution and pooling layers; these are the strides of the window and whether to use padding at the image edges. The strides parameter says how far the window should move at each step, and the padding parameter describes how we handle the pixels at the edges of the input.

With these two parameters, defining the two layers becomes:

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Conv2D(filters=64,

kernel\_size=3,

strides=1,

padding='same',

activation='relu'),

layers.MaxPool2D(pool\_size=2,

strides=1,

padding='same')

*# More layers follow*

])

Stride:

The distance the window moves at each step is called the *stride*. We need to specify the stride in both dimensions of the image; one for moving left to right and one for moving top to bottom. What effect does the stride have? Whenever the stride in either direction is greater than 1, the sliding window will skip over some of the pixels in the input at each step.

Because we want high-quality features to use for classification, convolutional layers will most often have strides=(1, 1). Increasing the stride means that we miss out on potentially valuble information in our summary. Maximum pooling layers, however, will almost always have stride values greater than 1, like (2, 2) or (3, 3), but not larger than the window itself.

Finally, note that when the value of the strides is the same number in both directions, you only need to set that number; for instance, instead of strides=(2, 2), you could use strides=2 for the parameter setting.

Padding:

When performing the sliding window computation, there is a question as to what to do at the boundaries of the input. Staying entirely inside the input image means the window will never sit squarely over these boundary pixels like it does for every other pixel in the input. Since we aren't treating all the pixels exactly the same, could there be a problem?

* What the convolution does with these boundary values is determined by its padding parameter. In TensorFlow, you have two choices; either padding='same' or padding='valid'. There are trade-offs with each.
* When we set padding='valid', the convolution window will stay entirely inside the input. The drawback is that the output shrinks , or losing its pixels, and shrinks more for larger kernels. This will limit the number of layers the network can contain, especially when inputs are small in size.
* The alternative is to use padding='same'. The trick here is to pad the input with 0's around its borders, using just enough 0's to make the size of the output the same as the size of the input. This can have the effect however of diluting the influence of pixels at the borders.

The VGG model we've been looking at uses same padding for all of its convolutional layers. Most modern *convnets* will use some combination of the two.

***Case study example: Exploring sliding windows***

To better understand the effect of the sliding window parameters, it can help to observe a feature extraction on a low-resolution image so that we can see the individual pixels. Let's just look at a simple circle. This next hidden cell will create an image and kernel for us.

import tensorflow as tf

import matplotlib.pyplot as plt

plt.rc('figure', autolayout=True)

plt.rc('axes', labelweight='bold', labelsize='large',

titleweight='bold', titlesize=18, titlepad=10)

plt.rc('image', cmap='magma')

image = circle([64, 64], val=1.0, r\_shrink=3)

image = tf.reshape(image, [\*image.shape, 1])

*# Bottom sobel*

kernel = tf.constant(

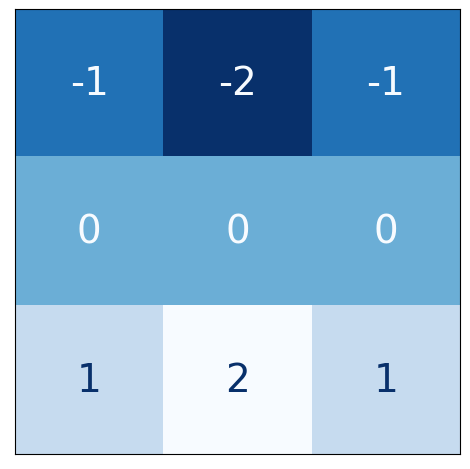
[[-1, -2, -1],

[0, 0, 0],

[1, 2, 1]],

)

show\_kernel(kernel)



The VGG architecture is fairly simple. It uses convolution with strides of 1 and maximum pooling with 2×2 windows and strides of 2. We've included a function in the visiontools utility script that will show us all the steps.

show\_extraction(

image, kernel,

*# Window parameters*

conv\_stride=1,

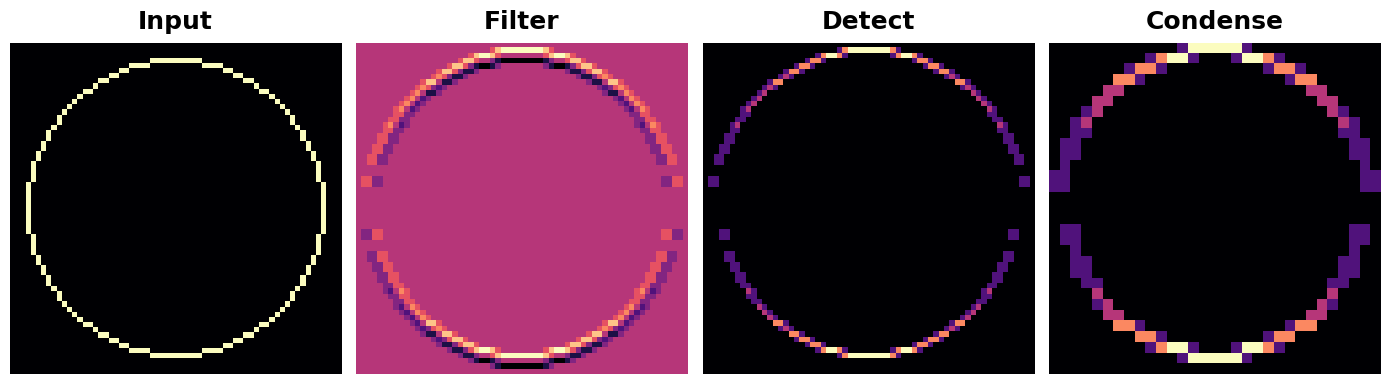
pool\_size=2,

pool\_stride=2,

subplot\_shape=(1, 4),

figsize=(14, 6),

)



And that works pretty well! The kernel was designed to detect horizontal lines, and we can see that in the resulting feature map the more horizontal parts of the input end up with the greatest activation. What would happen if we changed the strides of the convolution to 3?

show\_extraction(

image, kernel,

*# Window parameters*

conv\_stride=3,

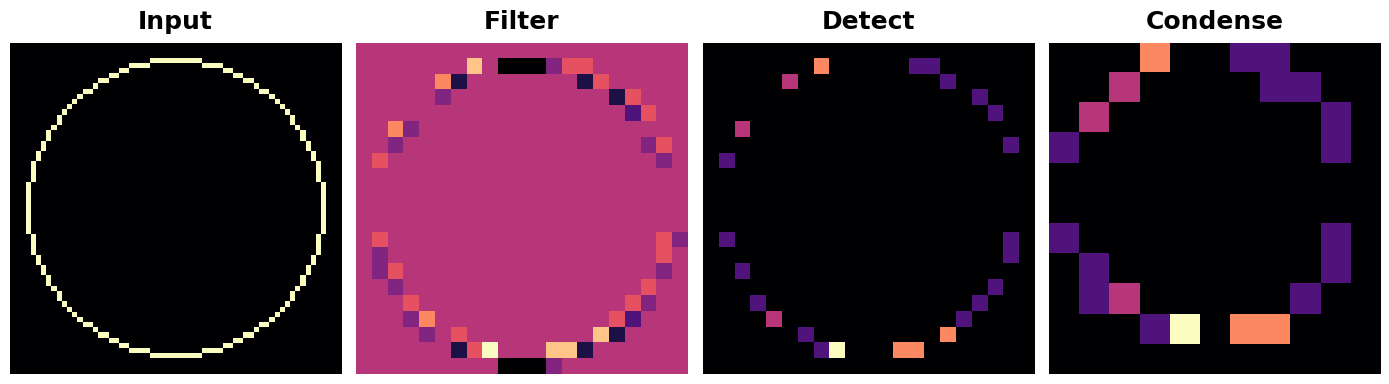
pool\_size=2,

pool\_stride=2,

subplot\_shape=(1, 4),

figsize=(14, 6),

)



This seems to reduce the quality of the feature extracted. Our input circle is rather "finely detailed," being only 1 pixel wide. A convolution with strides of 3 is too coarse to produce a good feature map from it.

Sometimes, a model will use a convolution with a larger stride in it's initial layer. This will usually be coupled with a larger kernel as well. The ResNet50 model, for instance, uses 7×7 kernels with strides of 2 in its first layer. This seems to accelerate the production of large-scale features without the sacrifice of too much information from the input.