We can compute the spatial size of the output volume as a function of the input volume size (W

), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by $(W-F+2P)/S+1$

. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output. Lets also see one more graphical example:

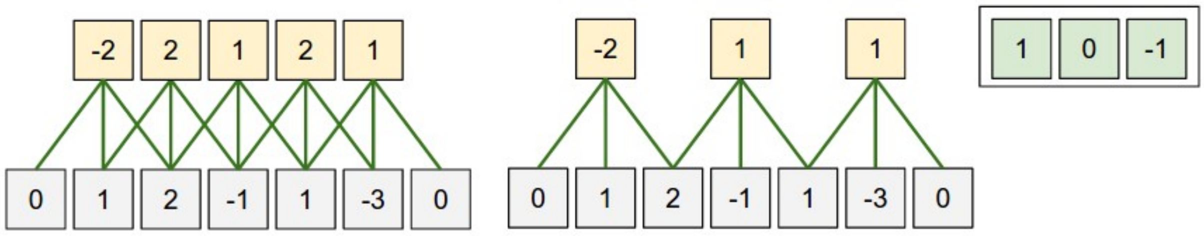


Illustration of spatial arrangement. In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. **Left:** The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. **Right:** The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3.

In general, setting zero padding to be $P=(F-1)/2$

when the stride is $S=1$
ensures that the input volume and output volume will have the same size spatially

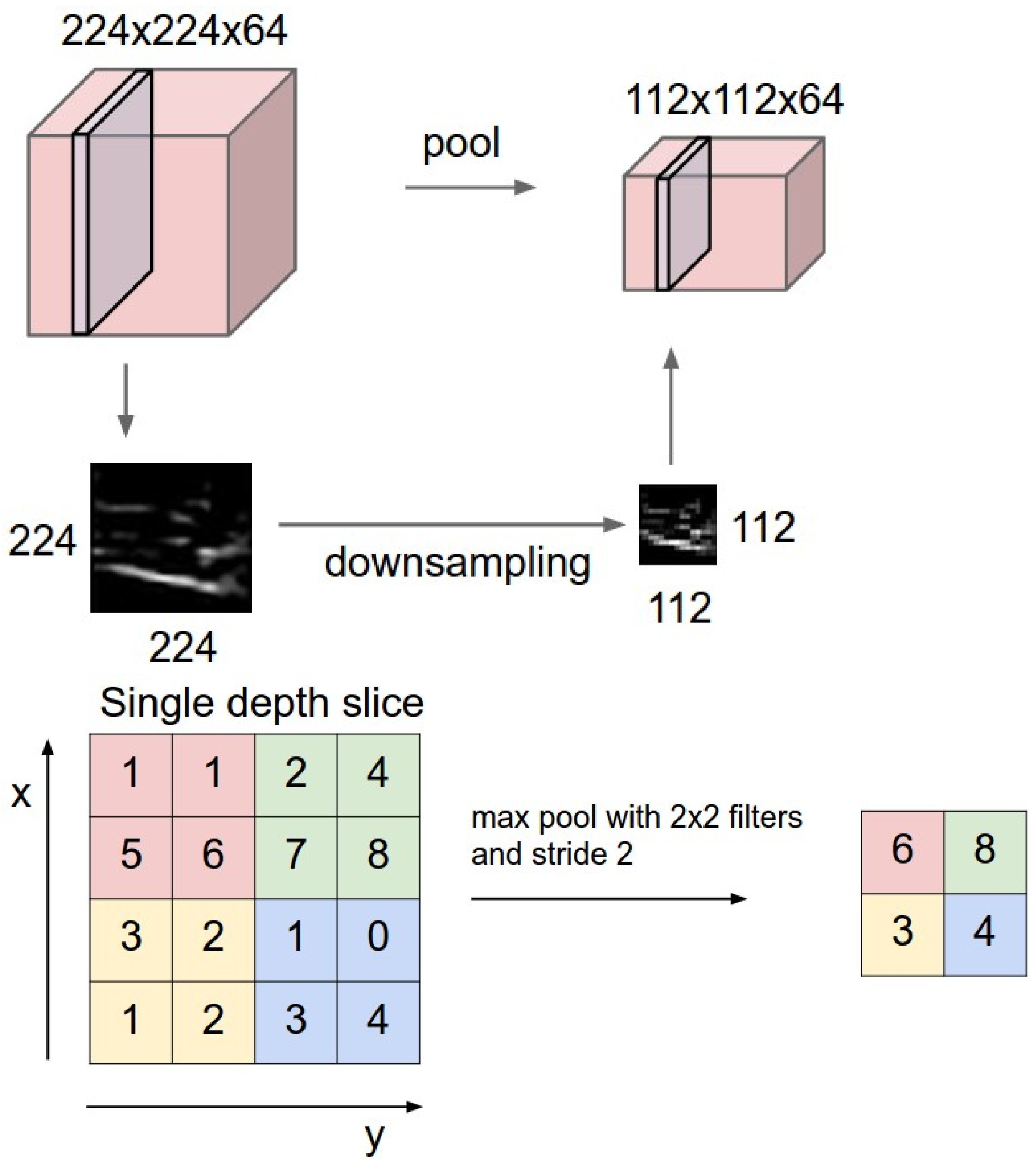
Constraints on strides. Note again that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size $W=10$

, no zero-padding is used $P=0$, and the filter size is $F=3$, then it would be impossible to use stride $S=2$, since $(W-F+2P)/S+1=(10-3+0)/2+1=4.5$

, i.e. not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a ConvNet library could throw an exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. As we will see in the ConvNet architectures section, sizing the ConvNets appropriately so that all the dimensions "work out" can be a real headache, which the use of zero-padding and some design guidelines will significantly alleviate.

Real-world example. The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $F=11$

, stride $S=4$ and no zero padding $P=0$. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of $K=96$



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Suppose you have an input space that is defined in one dimension. An example is (1 x 7 x 1). The seven columns in the input space represent seven pixel values, as follows: `[2, 4, 6, 1, 5, 3, 2]`.

Now, suppose that you want to convolve this input space using a (1 x 3) filter to perform some sort of feature detection. Suppose the filter has the following values: `[2, -1, 0.1]`.

Where should the convolving filter be positioned at the start? You can choose to start the convolution by aligning the left-most elements of the input space and the filter, as shown.

```
[2,  4,  6,  1, 5, 3, 2]
[2, -1, 0.1]
```

In this position, the dot product of the filter and the matrix is: `[2, -1, 0.1] * [2, 4, 6] = [(2*2), (-1*4), (0.1*6)] = [4, -4, 0.6]`. The result matrix `[4, -4, 0.6]` represents the values for the first pixel in the output array.

Now, you slide the filter to the right, by a stride equal to 1 pixel.

```
[2,  4,  6,  1,  5, 3, 2]
    [2, -1, 0.1]
```

Following the stride, the next dot product of the convolution is calculated: `[2, -1, 0.1] * [4, 6, 1] = [(2*4), (-1*6), (0.1*1)] = [8, -6, 0.1]`. The result `[8, -6, 0.1]` represents the values of the second pixel in the output matrix.

It follows that convolving the filter from the far left edge of the input space to the far right edge of the input space yields the following array of five output pixel values:

- `[2, -1, 0.1] * [2, 4, 6] = [4, -4, 0.6]` output pixel 1
- `[2, -1, 0.1] * [4, 6, 1] = [8, -6, 0.1]` output pixel 2
- `[2, -1, 0.1] * [6, 1, 5] = [12, -1, 0.5]` output pixel 3
- `[2, -1, 0.1] * [1, 5, 3] = [2, -5, 0.3]` output pixel 4
- `[2, -1, 0.1] * [5, 3, 2] = [10, -3, 0.2]` output pixel 5

Recall that the input space is a 1 x 7 image. If you convolve a 1 x 3 filter across a 1 x 7 input space, the matrix operations result in a 1 x 5 output space. Remember that it is desirable for the input and output images in a convolution layer to be the same size whenever possible.

Differently sized input and output images can be problematic for some algorithms. You need a way to generate output arrays that are the same size as the input arrays, given the size of the input space, the filter, and the stride.

You can use padding to meet these requirements. If you add a zero-valued pixel to both the left and right sides of the 1 x 7 input image matrix, you create an augmented, or “padded” 1 x 9 matrix. The padded 1 x 9 matrix still retains the original input image pixel values and geometry, but when convolved using a 1 x 3 filter, results in a 1 x 7 feature matrix that matches the dimensions of the original input matrix. The zero-value pixels facilitate the matrix computations that result in an output array with the desired dimensions, without disturbing the original input matrix values.

Now the augmented input matrix has nine pixel values as follows:

`[0, 2, 4, 6, 1, 5, 3, 2, 0].`

Convolving the 1 x 3 filter from left to right across the 1 x 9 input matrix results in the following output:

- `[2, -1, 0.1] * [0, 2, 4] = [0, -2, 0.4]` output pixel 1
- `[2, -1, 0.1] * [2, 4, 6] = [4, -4, 0.6]` output pixel 2
- `[2, -1, 0.1] * [4, 6, 1] = [8, -6, 0.1]` output pixel 3
- `[2, -1, 0.1] * [6, 1, 5] = [12, -1, 0.5]` output pixel 4
- `[2, -1, 0.1] * [1, 5, 3] = [2, -5, 0.3]` output pixel 5
- `[2, -1, 0.1] * [5, 3, 2] = [10, -3, 0.2]` output pixel 6
- `[2, -1, 0.1] * [3, 2, 0] = [6, -2, 0]` output pixel 7

Now we have input and output images of the same size.

However, suppose that you must use a new filter of dimensions (1 x 4) `[2, -1, 0.1, 5]`, instead of (1 x 3) `[2, -1, 0.1]`?

If you want to produce input and output matrices that are still the same size, you are going to need to add a third padding pixel to the input space. However, when your padding requires an odd number of elements, applying them to the matrix in a symmetrical manner becomes problematic.

Three padding pixels cannot be evenly divided between left and right sides of a matrix. One side of the matrix must have two padding pixels, while the opposing side will have only one. Suppose that you choose to pad the input matrix with one pixel on the left side, and two pixels on the right side, yielding an input image with ten pixels, as follows: `[0, 2, 4, 6, 1, 5, 3, 2, 0, 0]`

Convolving the 1 x 3 filter from left to right across the 1 x 10 input matrix results in the following output:

- `[2, -1, 0.1, 5] * [0, 2, 4, 6] = [0, -2, 0.4, 30]` output pixel 1
- `[2, -1, 0.1, 5] * [2, 4, 6, 1] = [4, -4, 0.6, 5]` output pixel 2
- `[2, -1, 0.1, 5] * [4, 6, 1, 5] = [8, -6, 0.1, 25]` output pixel 3
- `[2, -1, 0.1, 5] * [6, 1, 5, 3] = [12, -1, 0.5, 15]` output pixel 4
- `[2, -1, 0.1, 5] * [1, 5, 3, 2] = [2, -5, 0.3, 10]` output pixel 5
- `[2, -1, 0.1, 5] * [5, 3, 2, 0] = [10, -3, 0.2, 0]` output pixel 6
- `[2, -1, 0.1, 5] * [3, 2, 0, 0] = [6, -2, 0, 0]` output pixel 7

Consider a scenario where you have an odd image size, an even filter width, and a stride of 1:

```
Input = [1, 2, 3, 4, 5]
Filter = [1, 2]
Step = 1
```

Convolving a 1 x 5 input matrix with a 1 x 2 filter produces a 1 x 4 output matrix. You can use the corrective values `padLeft=0` and `padRight=1` to create the augmented (padded) input matrix `[1, 2, 3, 4, 5, 0]`.

The result of convolving the padded matrix with the 1 x 2 filter using CPU is as follows:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 5] = [4, 10]` output pixel 4
- `[1, 2] * [5, 0] = [5, 0]` output pixel 5

Now, look at the same scenario, but using GPU via the cuDNN library. The cuDNN library requires equal padding. If you use `padLeft=0` and `padRight=0`, then cuDNN generates only a 1 x 4 output matrix:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 5] = [4, 10]` output pixel 4

To satisfy the requirements of cuDNN, if you specify `padLeft = padRight = 1`, then the input matrix becomes 1 x 7, as `[0, 1, 2, 3, 4, 5, 0]`. A 1 x 7 input matrix convolved by a 1 x 2 filter generates a 1 x 6 output matrix as follows:

- `[1, 2] * [0, 1] = [0, 2]` output pixel 1
- `[1, 2] * [1, 2] = [1, 4]` output pixel 2
- `[1, 2] * [2, 3] = [2, 6]` output pixel 3
- `[1, 2] * [3, 4] = [3, 8]` output pixel 4
- `[1, 2] * [4, 5] = [4, 10]` output pixel 5
- `[1, 2] * [5, 0] = [5, 0]` output pixel 6

Likewise, there is a similar difference for cases where you have an even image size and an even filter size.

Consider a scenario where you have an image size of 1 x 4, a filter size of 1 x 2, and a stride of 1:

```
Input = [1, 2, 3, 4]
Filter = [1, 2]
Step = 1
```

Convolving a 1 x 4 input matrix with a 1 x 2 filter produces a 1 x 3 output matrix. The corrective values `padLeft=0` and `padRight=1` are used to create the augmented (padded) input matrix `[1, 2, 3, 4, 0]`.

The result of convolving the padded 1 x 5 matrix with the 1 x 2 filter (using CPU) generates a 1 x 4 output matrix, which is the same size as the original input matrix:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 0] = [4, 0]` output pixel 4

If you want to use GPU, and tell cuDNN that `padLeft = padRight = 0`, the output matrix is only 1 x 3:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3

On the other hand, if you want to use GPU, and tell cuDNN that `padLeft = padRight = 1`, the augmented input matrix is 1 x 6 `[0, 1, 2, 3, 4, 0]`, and the resulting output matrix is 1 x 5, which does not match the original input matrix dimensions:

- `[1, 2] * [0, 1] = [0, 2]` output pixel 1
- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 0] = [4, 0]` output pixel 4
- Padding will result in a "**same**" convolution.
- I talked about "stride", which is essentially how many pixels the filter shifts over the original image. Great, so now I can introduce the formula to quickly calculate the output size, knowing the filter size (f), stride (s), pad (p), and input size (n):
-

$$Output\ size = \left(\frac{n+2p-f}{s}+1\right) x \left(\frac{n+2p-f}{s}+1\right)$$

- Therefore, in general terms we have:
-

Input: $(n\,x\,n\,x\,n_c)$

Filter: $(f\,x\,f\,x\,n_c)$

Output: $\left(\left\lfloor\frac{n+2p-f}{s}+1\right\rfloor x \left\lfloor\frac{n+2p-f}{s}+1\right\rfloor x\,n_c'\right)$

- (with nc' as the number of filters, which are detecting different features)