An Illustrated Explanation of Performing 2D Convolutions Using Matrix Multiplications

Apr 6, 2019 · 6 min read ★

Introduction

In this article, I will explain how 2D Convolutions are implemented as matrix multiplications. This explanation is based on the notes of the CS231n Convolutional Neural Networks for Visual Recognition (Module 2). I assume the reader is familiar with the concept of a convolution operation in the context of a deep neural network. If not, this repo has a report and excellent animations explaining what convolutions are. The code to reproduce the computations in this article can be downloaded here.

Explanation

Small Example

Suppose we have a single channel 4 x 4 image, **X**, and its pixel values are as follows:

```
[ 1.,  2.,  3.,  4.]
[ 5.,  6.,  7.,  8.]
[ 9., 10., 11., 12.]
[13., 14., 15., 16.]
```

A single channel 4 x 4 image

Further suppose we define a 2D convolution with the following properties:

- kernel size: 2×2
- padding: 0
- stride: 1

- bias: 0
- output channels: 1
- initial weights, W: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Properties of the 2D convolution operation we want to perform on our image

This means that there will be 9 2 x 2 image patches that will be element-wise multiplied with the matrix W, like so:

Image Patch 1	Image Patch 2	Image Patch 3
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]
Image Patch 4	**Image Patch 5**	**Image Patch 6**
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]
Image Patch 7	**Image Patch 8**	**Image Patch 9**
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]

All the possible 2 x 2 image patches in **X** given the parameters of the 2D convolution. Each color represents a unique patch

These image patches can be represented as 4-dimensional column vectors and concatenated to form a single 4 x 9 matrix, **P**, like so:

```
[ 1.,  2.,  3.,  5.,  6.,  7.,  9., 10., 11.]
[ 2.,  3.,  4.,  6.,  7.,  8., 10., 11., 12.]
[ 5.,  6.,  7.,  9., 10., 11., 13., 14., 15.]
[ 6.,  7.,  8., 10., 11., 12., 14., 15., 16.]
```

The matrix of image patches, **P**

Notice that the i-th column of matrix **P** is actually the i-th image patch in column vector form.

The matrix of weights for the convolution layer, **W**, can be flattened to a 4-dimensional row vector , **K**, like so:

```
[1., 2., 3., 4.]
```

W as a flattened row vector **K**

To perform the convolution, we first matrix multiply **K** with **P** to get a 9-dimensional row vector (1 x 9 matrix) which gives us:

```
[ 44.,  54.,  64.,  84.,  94., 104., 124., 134., 144.]
```

The result of **K P**

Then we reshape the result of **K P** to the correct shape, which is a 3 x 3 x 1 matrix (channel dimension last). The channel dimension is 1 because we set the output filters to 1. The height and width is 3 because according to the CS231n notes:

$$\text{height} = \frac{\text{image height} - \text{kernel height} + 2(\text{number of 0 padding})}{\text{stride}} + 1$$

$$= \frac{4 - 2 + 2(0)}{1} + 1$$

$$= 3$$

Formula to compute the height after applying a convolution operation to an image. The formula to compute the width is the same (just replace image height and kernel height with image width and kernel width respectively

This means that the result of the convolution is:

```
[ 44.,  54.,  64.]
[ 84.,  94., 104.]
```

```
[124., 134., 144.]
```

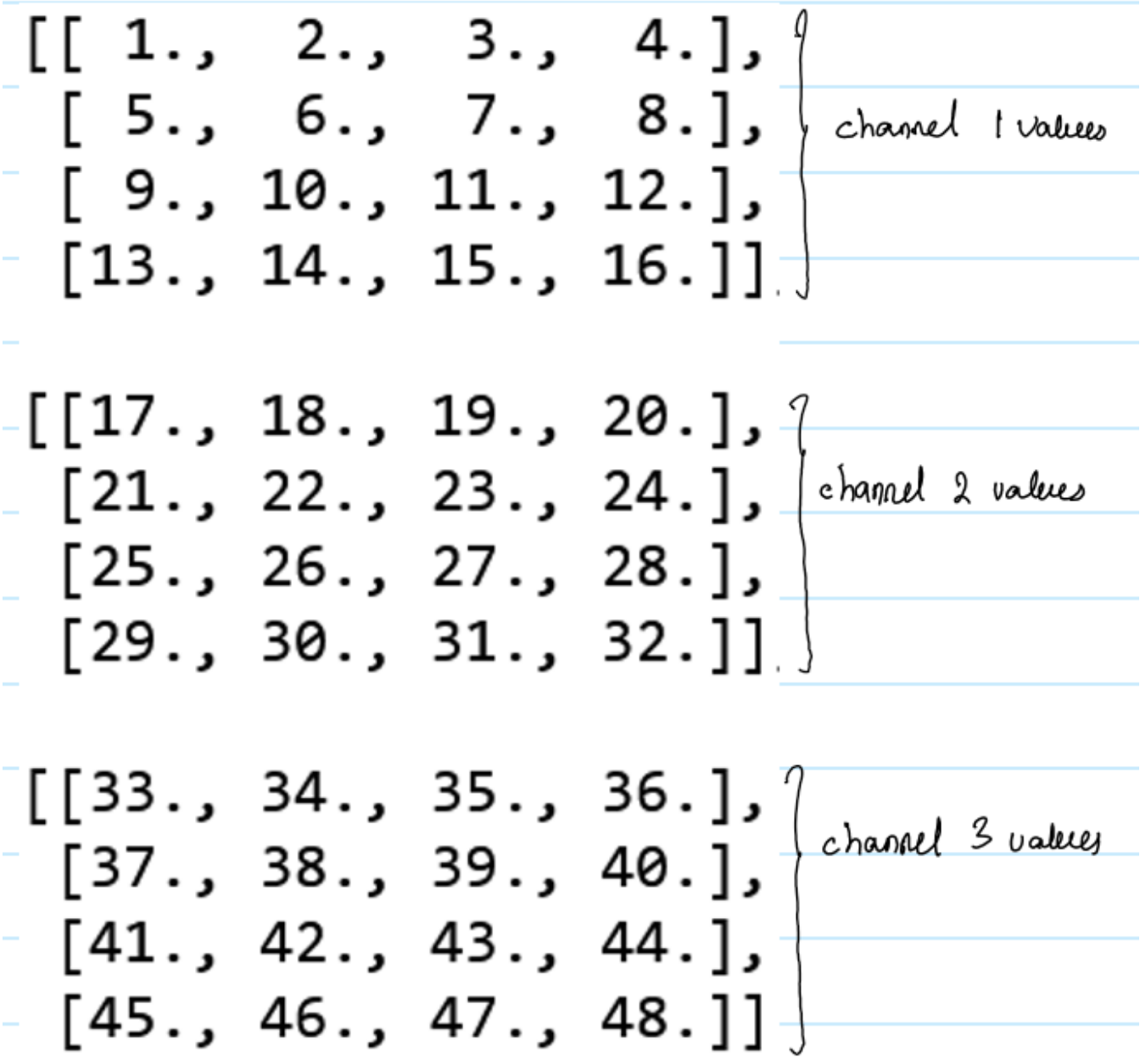
Final result of doing the convolution using matrix multiplications

Which checks out if we perform the convolution using PyTorch's built in functions (see this articles accompanying code for details).

Bigger Example

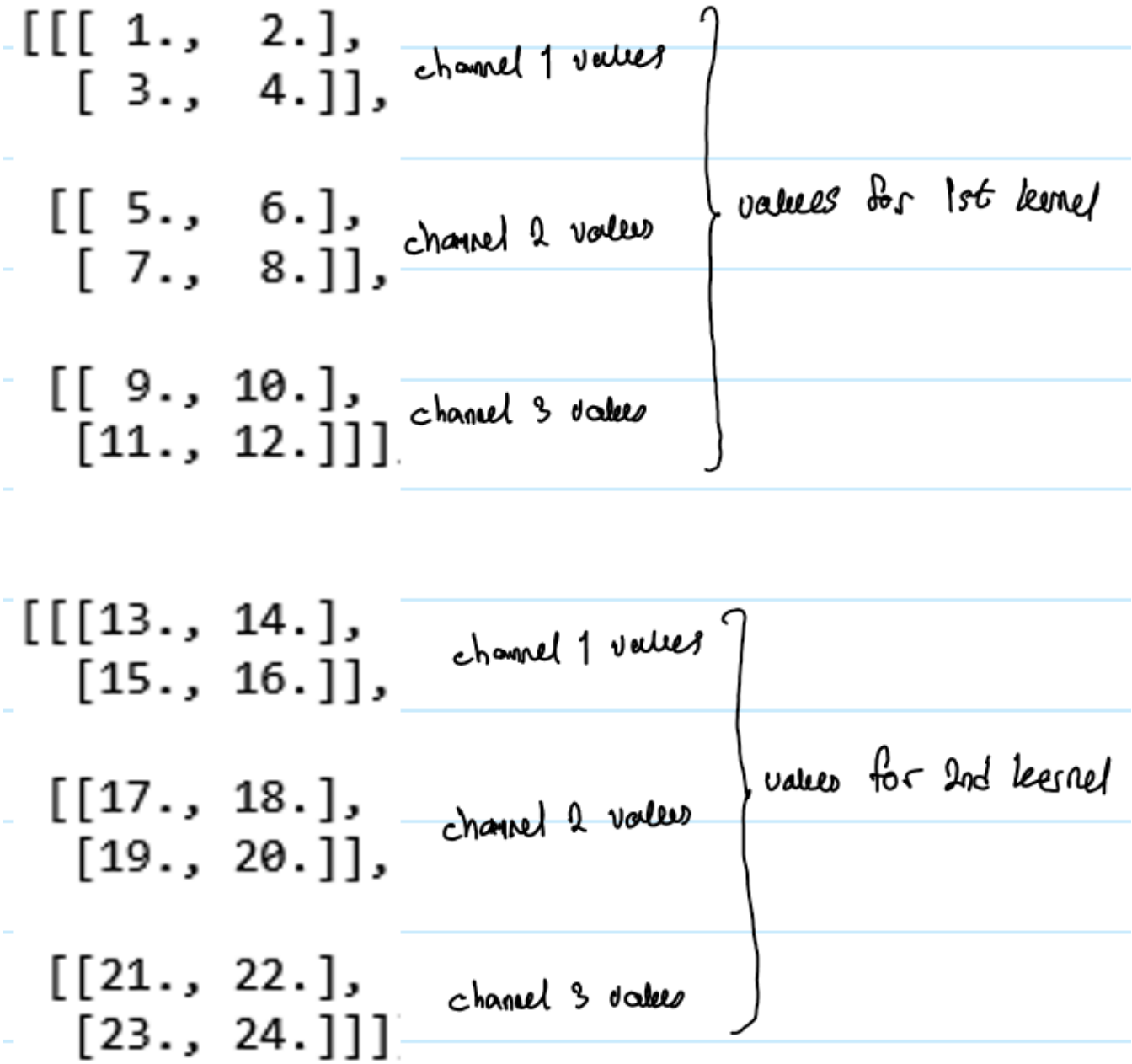
The example in the previous section assumes a single image and the output channels of the convolution is 1. What would change if we relax these assumptions?

Let's assume our input to the convolution is a 4 x 4 image with 3 channels with the following pixel values:



Pixel values in our 4 x 4 x 3 input image

As for our convolution, we will set it to have the same properties as the previous section except that its output filters is 2. This means that the initial weights matrix, **W**, must have shape (2, 2, 2, 3) i.e. (output filters, kernel height, kernel width, input image's channels). Let's set **W** to have the following values:



Values for **W** given a 2 x 2 kernel, 2 output filters and 3 channel input image

Notice that each output filters will have its own kernel (which is why we have 2 kernels in this example) and each kernel's channel is 3 (since the input image has 3 channels).

Since we are still convolving a 2 x 2 kernel on a 4 x 4 image with 0 zero-padding and stride 1, the number of image patches is still 9. However, the matrix of image patches, **P**, will be different. More specifically, the i-th column of **P** will be the concatenation of

the 1st, 2nd and 3rd channels values (as a column vector) corresponding to image patch i. **P** will now be a 12 x 9 matrix. The rows are 12 because each image patch has 3 channels and each channel has 4 elements since we set the kernel size to 2 x 2. Here's what **P** looks like:

```
[ 1.,  2.,  3.,  5.,  6.,  7.,  9., 10., 11.]
[ 2.,  3.,  4.,  6.,  7.,  8., 10., 11., 12.]
[ 5.,  6.,  7.,  9., 10., 11., 13., 14., 15.]
[ 6.,  7.,  8., 10., 11., 12., 14., 15., 16.]
[17., 18., 19., 21., 22., 23., 25., 26., 27.]
[18., 19., 20., 22., 23., 24., 26., 27., 28.]
[21., 22., 23., 25., 26., 27., 29., 30., 31.]
[22., 23., 24., 26., 27., 28., 30., 31., 32.]
[33., 34., 35., 37., 38., 39., 41., 42., 43.]
[34., 35., 36., 38., 39., 40., 42., 43., 44.]
[37., 38., 39., 41., 42., 43., 45., 46., 47.]
[38., 39., 40., 42., 43., 44., 46., 47., 48.]
```

The 12 x 9 matrix of image patches, **P**

As for **W**, each kernel will be flattened into a row vector and concatenated row-wise to form a 2 x 12 matrix, **K.** The i-th row of **K** is the concatenation of the 1st, 2nd and 3rd channel values (in row vector form) corresponding to the i-th kernel. Here's what **K** will look like:

```
[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]

[13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24.]
```

The 2 x 12 matrix of flattened kernel values in **W**, **K**

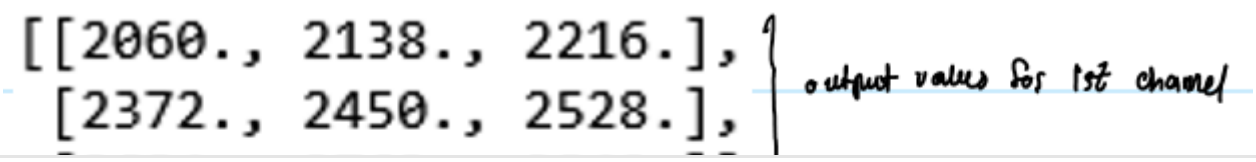
Now all that's left is to perform the matrix multiplication **K P** and reshape it to the correct shape. The correct shape is a 3 x 3 x 2 matrix (channel dimension last). Here's the result of the multiplication:

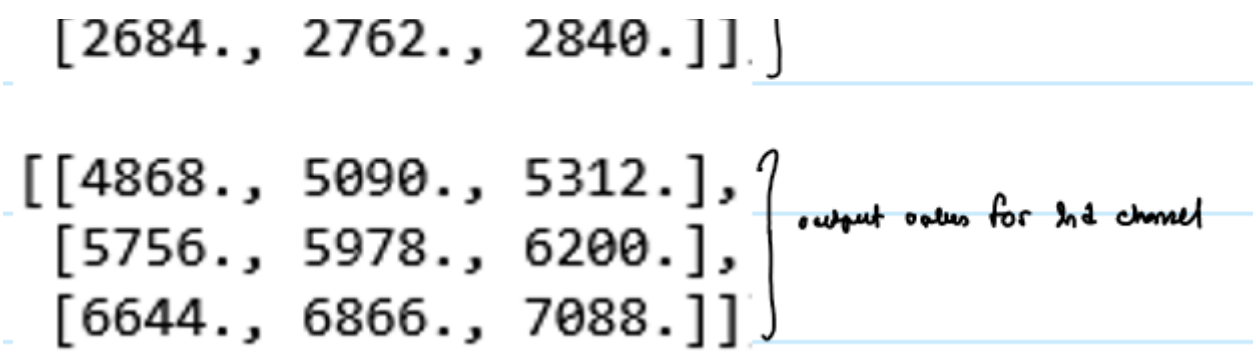
```
[2060., 2138., 2216., 2372., 2450., 2528., 2684., 2762., 2840.]

[4868., 5090., 5312., 5756., 5978., 6200., 6644., 6866., 7088.]
```

The result of **K P**, a 2 x 9 matrix

And here's the result after reshaping it to a 3 x 3 x 2 matrix:





Final result of doing the convolution using matrix multiplications

Which again checks out if we to perform the convolution using PyTorch's built in functions (see this article's accompanying code for details).

So What?

Why should we care about expressing 2D convolutions in terms of matrix multiplications? Besides having an efficient implementation suitable for running on a GPU, knowledge of this approach will allow us to reason about the behavior of a deep convolutional neural network. For example, He et. al. (2015) expressed 2D convolutions in terms of matrix multiplications which allowed them to apply the properties of random matrices/vectors to argue for a better weights initialization routine.

Conclusion

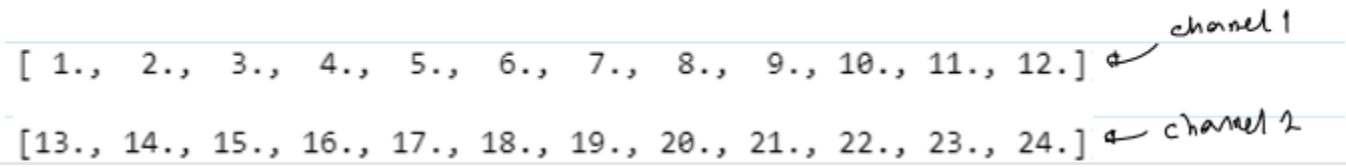
In this article, I have explained how to perform 2D convolutions using matrix multiplications by walking through two small examples. I hope this is sufficient for you to generalize to arbitrary input image dimensions and convolution properties. Let me know in the comments if anything is unclear.

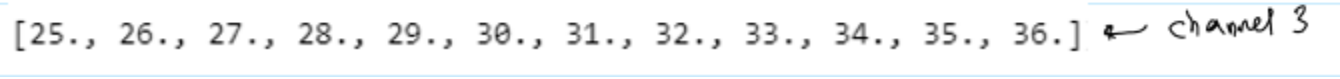
Appendix

1D Convolution

The method described in this article generalizes to 1D convolutions as well.

For example, suppose your input is a 3 channel 12-dimensional vector like so:



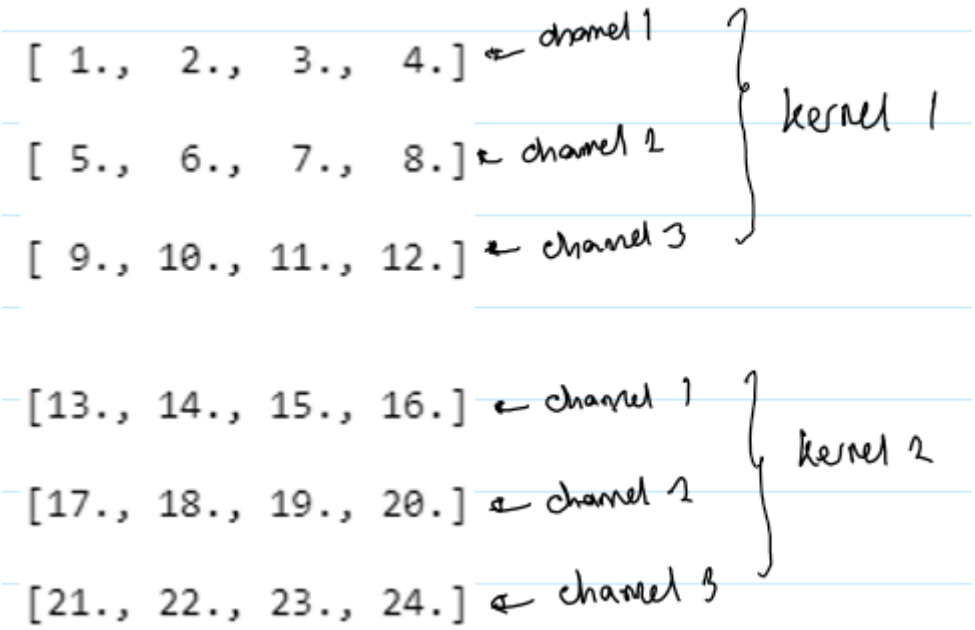


An input that is a 12-D vector with 3 channels

If we set our 1D convolution to have the following parameters:

- kernel size: 1 x 4
- output channels: 2
- stride: 2
- padding: 0
- bias: 0

Then the parameters in the convolution operation, **W**, will be a tensor with shape (2 , 3, 1, 4). Let's set **W** to have the following values:



Values for **W**. Note that each kernel is a (3, 1, 4) tensor

Based on the parameters of the convolution operation, the matrix of "image" patches **P**, will have a shape (12, 5) (5 image patches where each image patch is a 12-D vector since a patch has 4 elements across 3 channels) and will look like this:

```
[ 1.,  3.,  5.,  7.,  9.]
[ 2.,  4.,  6.,  8., 10.]
[ 3.,  5.,  7.,  9., 11.]
[ 4.,  6.,  8., 10., 12.]
```

```
[13., 15., 17., 19., 21.]
[14., 16., 18., 20., 22.]
[15., 17., 19., 21., 23.]
[16., 18., 20., 22., 24.]
[25., 27., 29., 31., 33.]
[26., 28., 30., 32., 34.]
[27., 29., 31., 33., 35.]
[28., 30., 32., 34., 36.]
```

Values for **P**, a 12 x 5 matrix

Next, we flatten **W** to get **K**, which has shape (2, 12) since there are 2 kernels and each kernel has 12 elements. This is what **K** looks like:

```
[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.]
[13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24.]
```

Values of **K**, a 2 x 12 matrix

Now we can multiply **K** with **P** which gives:

```
[1530., 1686., 1842., 1998., 2154.]
[3618., 4062., 4506., 4950., 5394.]
```

The result of **K P**, a 2 x 5 matrix

Finally, we reshape the **K P** to the correct shape, which according to the formula is an "image" with shape (1, 5). This means the result of this convolution is a tensor with shape (2, 1, 5) since we set the output channels to 2. This is what the final result looks like:

```
[[[1530., 1686., 1842., 1998., 2154.]],
 [[3618., 4062., 4506., 4950., 5394.]]]
```


The result of **K P** after reshaping to the correct shape

Which as expected, checks out if we were to perform the convolution using PyTorch's built-in functions.

References

A guide to convolution arithmetic for deep learning; Dumoulin and Visin. 2018

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification; He et. al. 2015

Machine Learning | Deep Learning | Neural Networks | Convolutional Network | Artificial Intelligence