



Open in app

Get started



Published in Towards Data Science



Akshaj Verma

Follow

Jan 18, 2020 · 9 min read · Listen



Save



How to train you neural net [Image [0]]

HOW TO TRAIN YOUR NEURAL NET

Pytorch [Basics] — Intro to CNN

This blog post takes you through the different types of CNN operations in PyTorch.

In this blog post, we will implement 1D and 2D convolutions using `torch.nn`.



[Open in app](#)[Get started](#)

such as audio, time series, and NLP. Convolution is one of the main building blocks of a CNN. The term convolution refers to the mathematical combination of two functions to produce a third function. It merges two sets of information.

We won't go over a lot of theory here. There's plenty of fantastic material available online for this.

Types of CNN operations

CNNs are majorly used for applications surrounding images, audio, videos, text, and time series modelling. There are 3 types of convolution operations.

- **1D convolution** — majorly used where the input is sequential such as text or audio.
- **2D convolution** — majorly used where the input is an image.
- **3D convolution** — majorly used in 3D medical imaging or detecting events in videos. This is outside the scope of this blog post. We will only focus on the first two.

1D Convolution for 1D Input

The filter slides along a single dimension to produce an output. The following diagrams are taken from [this Stackoverflow answer](#).



1D Convolution for 1D Input [Image [1] [credits](#)]

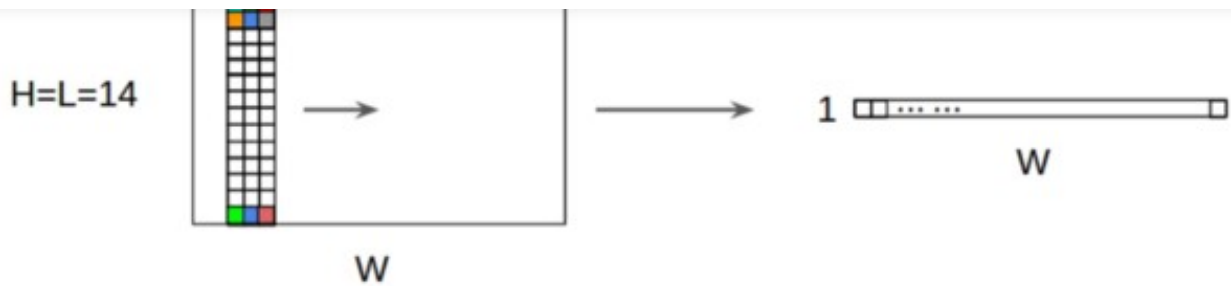
1D Convolution for 2D Input





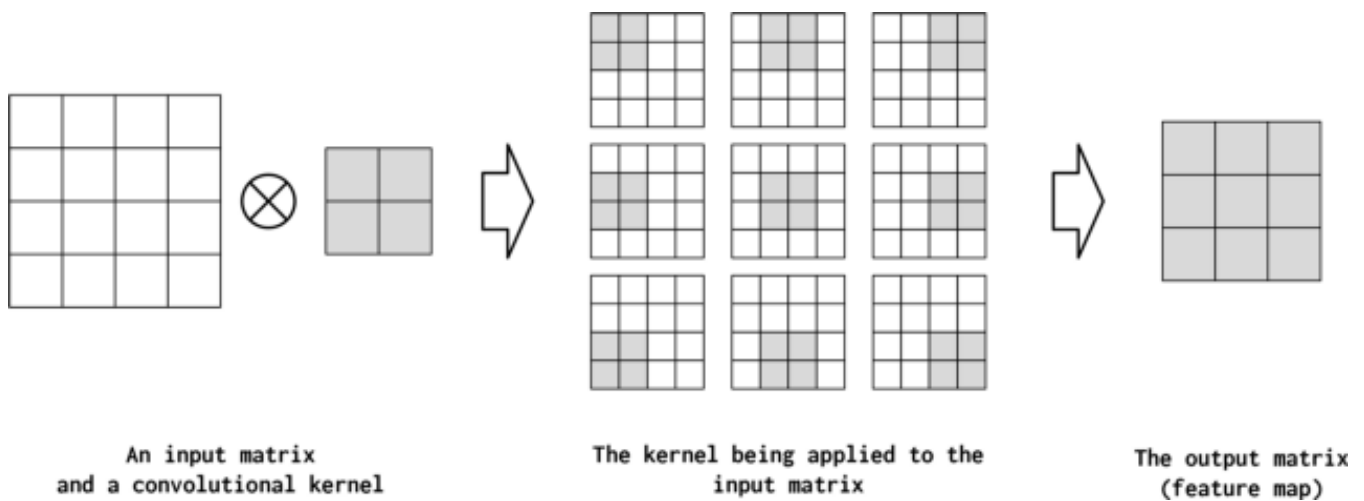
Open in app

Get started



1D Convolution for 2D Input [Image [2] [credits](#)]

2D Convolution for 2D Input



2D Convolution for 2D Input [Image [3] [credits](#)]

Check out this [Stackoverflow answer](#) for more information on different types of CNN operations.

A Few Key Terminologies

The terminologies are explained for 2D convolutions and 2D inputs ie. images because I could not find relevant visualizations for 1D Convolutions. All the visualizations are taken from [here](#).

Convolution Operation

To calculate the output dimension after a convolution operation, we can use the following formula.



[Open in app](#)[Get started](#)

n_{in} : number of input features
 n_{out} : number of output features
 k : convolution kernel size
 p : convolution padding size
 s : convolution stride size

Convolution Output Formula [Image [4]]

The kernel/filter slides over the input signal as shown below. You can see the **filter** (the green square) is sliding over our **input** (the blue square) and the sum of the convolution goes into the **feature map** (the red square).

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

Convolution Operation [Image [5]]

Filter/Kernel

A convolution is performed on an input image using filters. The output of convolution is known as a feature map.



[Open in app](#)[Get started](#)

0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

Filter [Image [6]]

In CNN terminology, the 3×3 matrix is called a '**filter**' or '**kernel**' or '**feature detector**' and the matrix formed by sliding the filter over the image and computing the dot product is called the '**Convolved Feature**' or '**Activation Map**' or the '**Feature Map**'. It is important to note that filters act as feature detectors from the original input image.

more filters = more feature maps = more features.








A filter is nothing but a matrix of numbers. Following are the different types of filters —





Open in app

Get started

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Different types of filters [Image [7]]

Stride

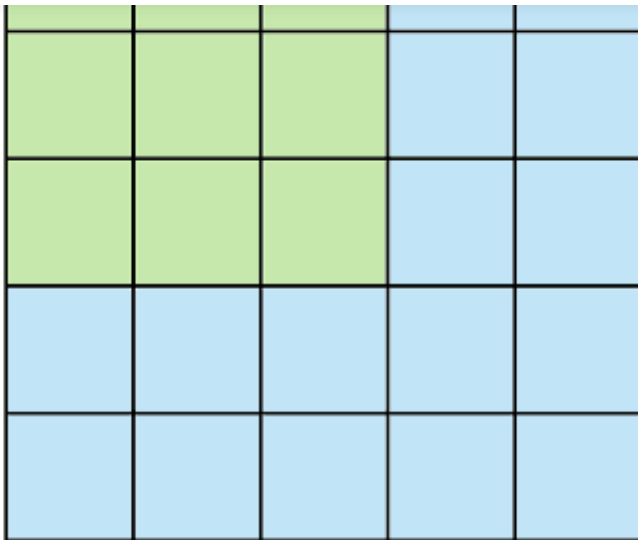
Stride specifies how much we move the convolution filter at each step.



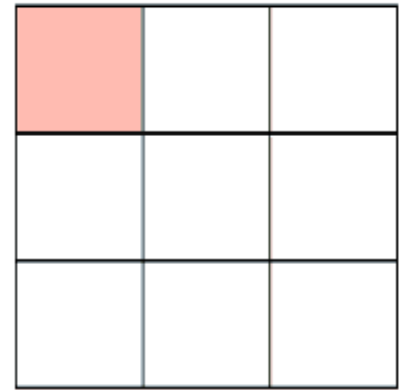


Open in app

Get started



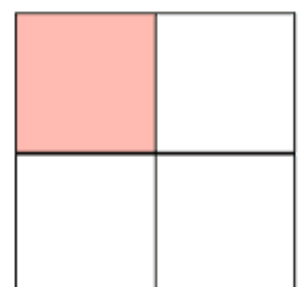
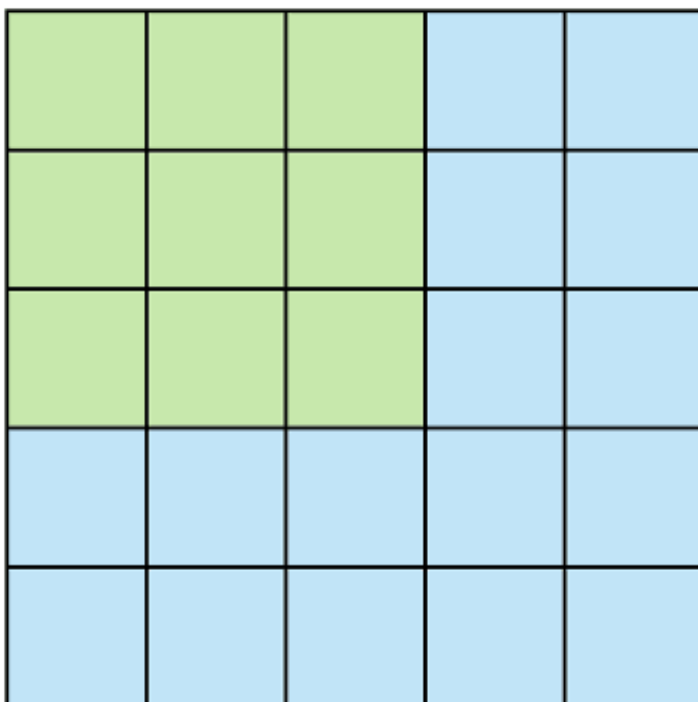
Stride 1



Feature Map

Stride of 1 [Image [8]]

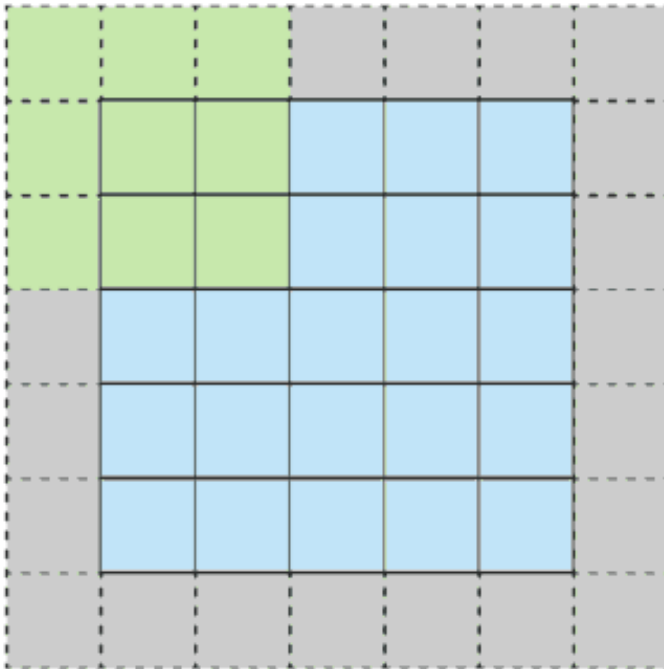
We can have bigger strides if we want less overlap between the receptive fields. This also makes the resulting feature map smaller since we are skipping over potential locations. The following figure demonstrates a stride of 2. Note that the feature map got smaller.



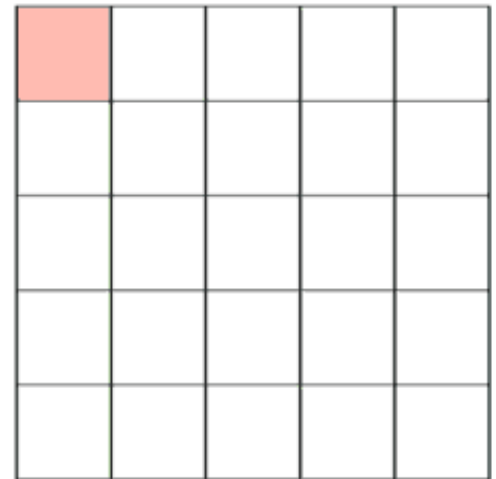
[Open in app](#)[Get started](#)

Padding

Here we have retained more information from the borders and have also preserved the size of the image.



Stride 1 with Padding



Feature Map

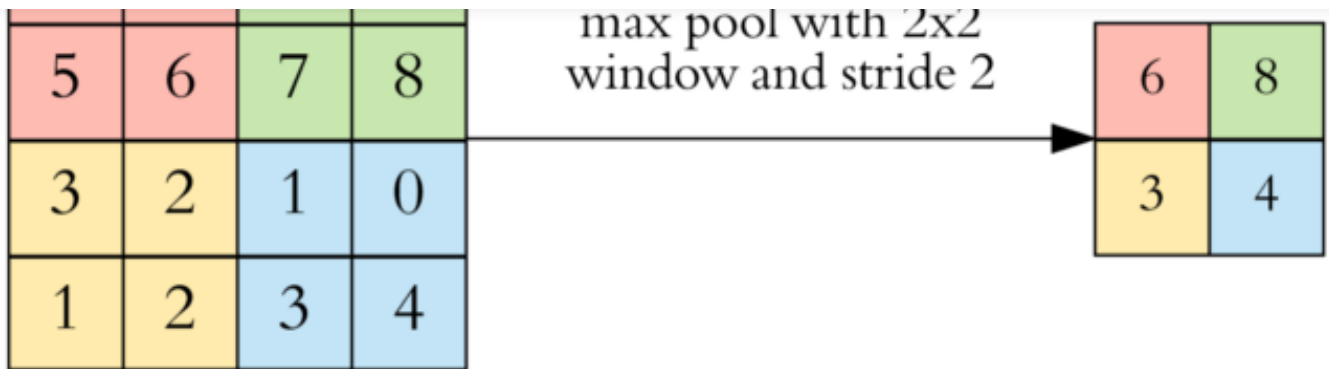
Padding [Image [10]]

We see that the size of the feature map is smaller than the input, because the convolution filter needs to be contained in the input. If we want to maintain the same dimensionality, we can use *padding* to surround the input with zeros.

Pooling

We apply pooling to reduce dimensionality.



[Open in app](#)[Get started](#)

Max Pooling [Image [11]]

- Pooling reduces the size of the input and makes the feature dimension smaller.
- Because of lower spatial size, the number of parameters in the network are reduced. This helps in combating overfitting.
- Pooling makes the network robust to distortions in the image because we take the aggregate(max, sum, average etc.) of the pixel values in a neighborhood.

Import Libraries

```
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

Input Data

To start with, we define a few input tensors which we will use throughout this blog post.

`input_1d` is a 1 dimensional float tensor. `input_2d` is a 2 dimensional float tensor.
`input_2d_img` is a 3 dimensional float tensor which represents an image.

```
input_1d = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype =  
torch.float)
```



[Open in app](#)[Get started](#)

```
3, 4, 5, 6, 7, 8, 9, 10]], [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]], dtype = torch.float)
```

```
##### OUTPUT #####
```

Input 1D:

```
input_1d.shape: torch.Size([10])
```

input_1d:

```
tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

Input 2D:

```
input_2d.shape: torch.Size([2, 5])
```

input_2d:

```
tensor([[ 1.,  2.,  3.,  4.,  5.],
        [ 6.,  7.,  8.,  9., 10.]])
```

input_2d_img:

```
input_2d_img.shape: torch.Size([3, 3, 10])
```

input_2d_img:

```
tensor([[[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.],
          [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.],
          [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]],

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

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

1D Convolution

`nn.Conv1d()` applies 1D convolution over the input. `nn.Conv1d()` expects the input to be of the shape `[batch_size, input_channels, signal_length]`.

You can check out the complete list of parameters in the official [PyTorch Docs](#). The required parameters are —

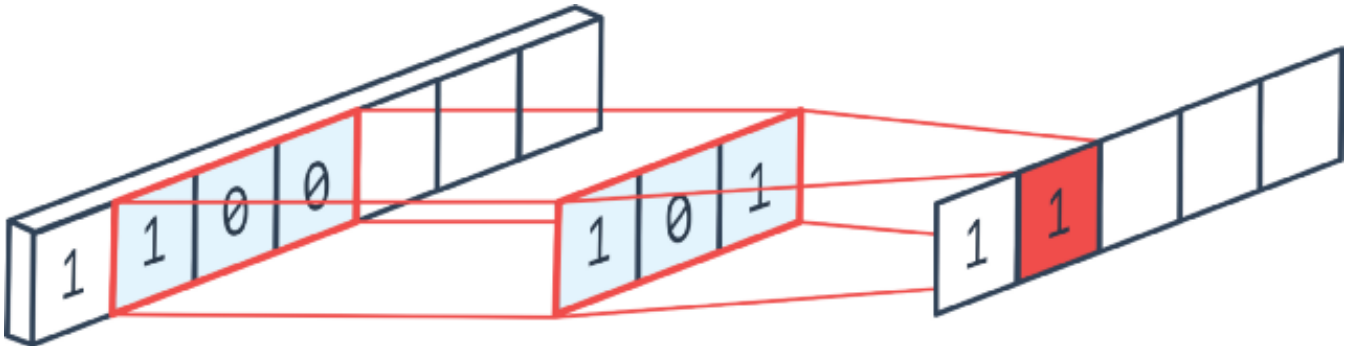
- **in_channels** (*python:int*) — Number of channels in the input signal. This should



[Open in app](#)[Get started](#)

- `kernel_size` (*python.int or tuple*) — size of the convolving kernel.

Conv1d — Input 1d



Conv1d-Input1d Example [Image [12] [credits](#)]

The input is a 1D signal which consists of 10 numbers. We will convert this into a tensor of size [1, 1, 10].

```
input_1d = input_1d.unsqueeze(0).unsqueeze(0)
input_1d.shape

##### OUTPUT #####

torch.Size([1, 1, 10])
```

CNN Output with `out_channels=1`, `kernel_size=3` and `stride=1` .

```
cnn1d_1 = nn.Conv1d(in_channels=1, out_channels=1, kernel_size=3,
stride=1)

print("cnn1d_1: \n")
print(cnn1d_1(input_1d).shape, "\n")
print(cnn1d_1(input_1d))

##### OUTPUT #####

cnn1d_1:

torch.Size([1, 1, 8])

tensor([[[[-1.2353, -1.4051, -1.5749, -1.7447, -1.9145, -2.0843,
-2.2541, -2.4239]]], grad_fn=<SqueezeBackward1>)
```



[Open in app](#)[Get started](#)

```
cnn1d_2 = nn.Conv1d(in_channels=1, out_channels=1, kernel_size=3,
stride=2)

print("cnn1d_2: \n")
print(cnn1d_2(input_1d).shape, "\n")
print(cnn1d_2(input_1d))

##### OUTPUT #####

cnn1d_2:

torch.Size([1, 1, 4])

tensor([[[[0.5107, 0.3528, 0.1948, 0.0368]]], grad_fn=
<SqueezeBackward1>)
```

CNN Output with `out_channels=1`, `kernel_size=2` and `stride=1` .

```
cnn1d_3 = nn.Conv1d(in_channels=1, out_channels=1, kernel_size=2,
stride=1)

print("cnn1d_3: \n")
print(cnn1d_3(input_1d).shape, "\n")
print(cnn1d_3(input_1d))

##### OUTPUT #####

cnn1d_3:

torch.Size([1, 1, 9])

tensor([[[[0.0978, 0.2221, 0.3465, 0.4708, 0.5952, 0.7196, 0.8439,
0.9683, 1.0926]]], grad_fn=<SqueezeBackward1>)
```

CNN Output with `out_channels=5`, `kernel_size=3` and `stride=2` .

```
cnn1d_4 = nn.Conv1d(in_channels=1, out_channels=5, kernel_size=3,
stride=1)

print("cnn1d_4: \n")
print(cnn1d_4(input_1d).shape, "\n")
print(cnn1d_4(input_1d))

##### OUTPUT #####
```



[Open in app](#)[Get started](#)

```
-6.0307e+00, -7.0781e+00, -8.1255e+00, -9.1729e+00],  
      [-4.6073e-02, -3.4436e-02, -2.2799e-02, -1.1162e-02,  
4.7439e-04, 1.2111e-02, 2.3748e-02, 3.5385e-02],  
      [-1.5541e+00, -1.8505e+00, -2.1469e+00, -2.4433e+00,  
-2.7397e+00, -3.0361e+00, -3.3325e+00, -3.6289e+00],  
      [ 6.6593e-01, 1.2362e+00, 1.8066e+00, 2.3769e+00,  
2.9472e+00, 3.5175e+00, 4.0878e+00, 4.6581e+00],  
      [ 2.0414e-01, 6.0421e-01, 1.0043e+00, 1.4044e+00,  
1.8044e+00, 2.2045e+00, 2.6046e+00, 3.0046e+00]]],  
grad_fn=<SqueezeBackward1>)
```

Conv1d — Input 2d

To apply 1D convolution on a 2d input signal, we can do the following. First, we define our input tensor of the size [1, 2, 5] where `batch_size = 1`, `input_channels = 2`, and `signal_length = 5`.

```
input_2d = input_2d.unsqueeze(0)  
input_2d.shape  
  
##### OUTPUT #####  
  
torch.Size([1, 2, 5])
```

CNN Output with `in_channels=2`, `out_channels=1`, `kernel_size=3`, `stride=1`.

```
cnn1d_5 = nn.Conv1d(in_channels=2, out_channels=1, kernel_size=3,  
stride=1)  
  
print("cnn1d_5: \n")  
print(cnn1d_5(input_2d).shape, "\n")  
print(cnn1d_5(input_2d))  
  
##### OUTPUT #####  
  
cnn1d_5:  
  
torch.Size([1, 1, 3])  
  
tensor([[[[-6.6836, -7.6893, -8.6950]]]], grad_fn=<SqueezeBackward1>)
```

CNN Output with `in_channels=2`, `out_channels=1`, `kernel_size=3`, `stride=2`.



[Open in app](#)[Get started](#)

```
print("cnn1d_6: \n")
print(cnn1d_6(input_2d).shape, "\n")
print(cnn1d_6(input_2d))

##### OUTPUT #####

cnn1d_6:
torch.Size([1, 1, 2])
tensor([[[[-3.4744, -3.7142]]], grad_fn=<SqueezeBackward1>)
```



433



2

CNN Output with `in_channels=2`, `out_channels=1`, `kernel_size=2`, `stride=1` .

```
cnn1d_7 = nn.Conv1d(in_channels=2, out_channels=1, kernel_size=2,
stride=1)

print("cnn1d_7: \n")
print(cnn1d_7(input_2d).shape, "\n")
print(cnn1d_7(input_2d))

##### OUTPUT #####

cnn1d_7:
torch.Size([1, 1, 4])
tensor([[[[0.5619, 0.6910, 0.8201, 0.9492]]], grad_fn=
<SqueezeBackward1>)
```

CNN Output with `in_channels=2`, `out_channels=5`, `kernel_size=3`, `stride=1` .

```
cnn1d_8 = nn.Conv1d(in_channels=2, out_channels=5, kernel_size=3,
stride=1)

print("cnn1d_8: \n")
print(cnn1d_8(input_2d).shape, "\n")
print(cnn1d_8(input_2d))

##### OUTPUT #####

cnn1d_8:
torch.Size([1, 5, 3])
tensor([[[[ 1.5024,  2.4199,  3.3373],
           [ 0.2980, -0.0873, -0.4727],
           [ 0.1111,  0.1111,  0.1111],
           [ 0.1111,  0.1111,  0.1111],
           [ 0.1111,  0.1111,  0.1111]]], grad_fn=
```





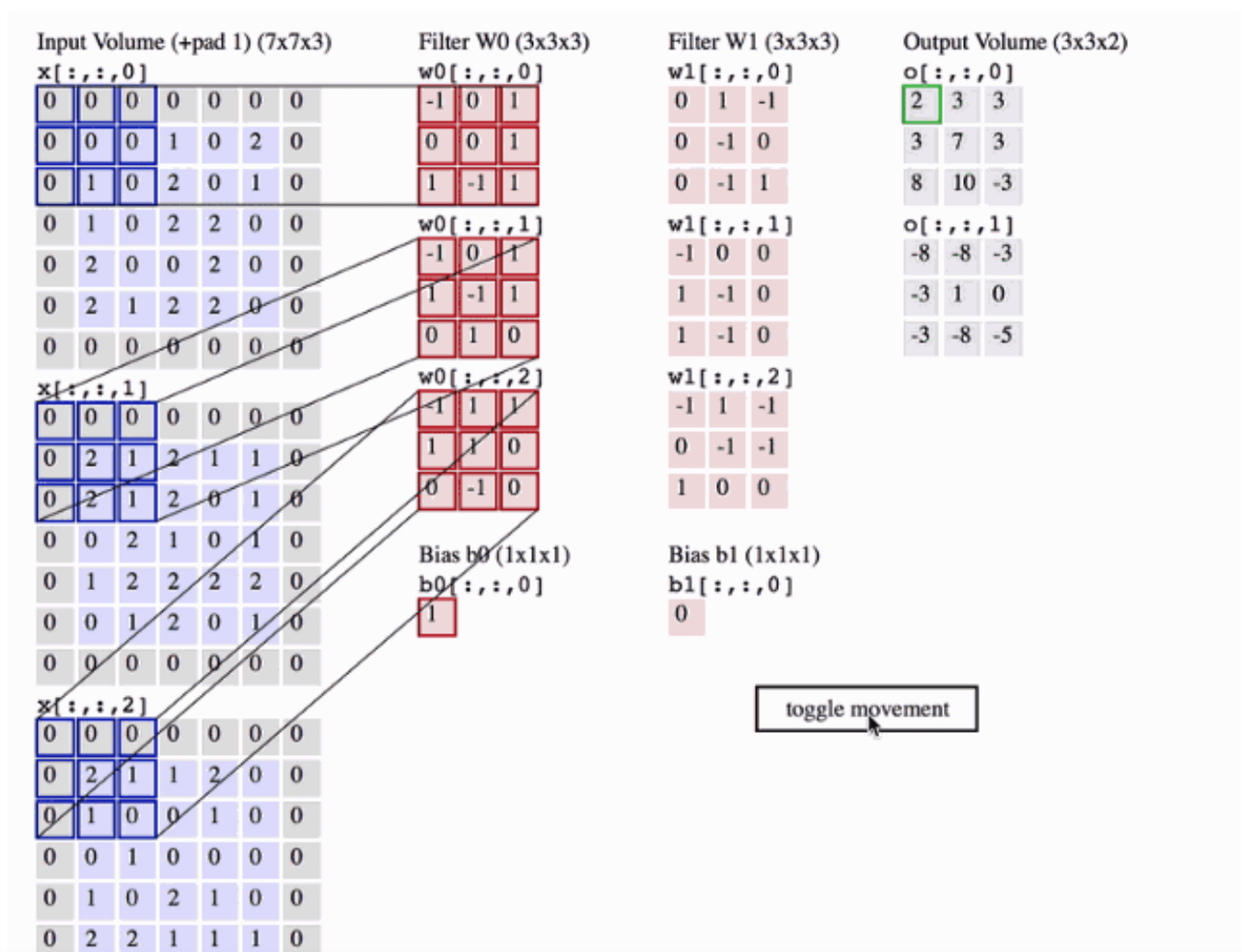
2D Convolution

`nn.Conv2d()` applies 2D convolution over the input. `nn.Conv2d()` expects the input to be of the shape `[batch_size, input_channels, input_height, input_width]`.

You can check out the complete list of parameters in the official [PyTorch Docs](#). The required parameters are —

- **in_channels** (*python:int*) — Number of channels in the 2d input eg. image.
- **out_channels** (*python:int*) — Number of channels produced by the convolution.
- **kernel_size** (*python:int or tuple*) — Size of the convolving kernel

Conv2d — Input 2d



[Open in app](#)[Get started](#)

To apply 2D convolution on a 2d input signal (eg. images), we can do the following.

First, we define our input tensor of the size [1, 3, 3, 10] where `batch_size = 1`,

`input_channels = 3`, `input_height = 3`, and `input_width = 10`.

```
input_2d_img = input_2d_img.unsqueeze(0)
input_2d_img.shape

##### OUTPUT #####

torch.Size([1, 3, 3, 10])
```

CNN Output with `in_channels=3`, `out_channels=1`, `kernel_size=3`, `stride=1`.

```
cnn2d_1 = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3,
stride=1)

print("cnn2d_1: \n")
print(cnn2d_1(input_2d_img).shape, "\n")
print(cnn2d_1(input_2d_img))

##### OUTPUT #####

cnn2d_1:

torch.Size([1, 1, 1, 8])

tensor([[[[-1.0716, -1.5742, -2.0768, -2.5793, -3.0819, -3.5844,
-4.0870, -4.5896]]]], grad_fn=<MkldnnConvolutionBackward>)
```

CNN Output with `in_channels=3`, `out_channels=1`, `kernel_size=3`, `stride=2`.

```
cnn2d_2 = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3,
stride=2)

print("cnn2d_2: \n")
print(cnn2d_2(input_2d_img).shape, "\n")
print(cnn2d_2(input_2d_img))

##### OUTPUT #####

cnn2d_2:

torch.Size([1, 1, 1, 4])
```



[Open in app](#)[Get started](#)

CNN Output with `in_channels=3, out_channels=1, kernel_size=2, stride=1` .

```
cnn2d_3 = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=2,
stride=1)

print("cnn2d_3: \n")
print(cnn2d_3(input_2d_img).shape, "\n")
print(cnn2d_3(input_2d_img))

##### OUTPUT #####

cnn2d_3:
torch.Size([1, 1, 2, 9])

tensor([[[[-0.8046, -1.5066, -2.2086, -2.9107, -3.6127, -4.3147,
-5.0167, -5.7188, -6.4208],
          [-0.8046, -1.5066, -2.2086, -2.9107, -3.6127, -4.3147,
-5.0167, -5.7188, -6.4208]]]], grad_fn=<MkldnnConvolutionBackward>)
```

CNN Output with `in_channels=3, out_channels=5, kernel_size=3, stride=1` .

```
cnn2d_4 = nn.Conv2d(in_channels=3, out_channels=5, kernel_size=3,
stride=1)

print("cnn2d_4: \n")
print(cnn2d_4(input_2d_img).shape, "\n")
print(cnn2d_4(input_2d_img))

##### OUTPUT #####

cnn2d_4:
torch.Size([1, 5, 1, 8])

tensor([[[[-2.0868e+00, -2.7669e+00, -3.4470e+00, -4.1271e+00,
-4.8072e+00, -5.4873e+00, -6.1673e+00, -6.8474e+00]],
          [[-4.5052e-01, -5.5917e-01, -6.6783e-01, -7.7648e-01,
-8.8514e-01, -9.9380e-01, -1.1025e+00, -1.2111e+00]],
          [[ 6.6228e-01,  8.3826e-01,  1.0142e+00,  1.1902e+00,
 1.3662e+00, 1.5422e+00,  1.7181e+00,  1.8941e+00]],
          [[-5.4425e-01, -1.2149e+00, -1.8855e+00, -2.5561e+00,
-3.2267e+00, -3.8973e+00, -4.5679e+00, -5.2385e+00]]],
          ]])
```





Open in app

Get started

Thank you for reading. Suggestions and constructive criticism are welcome. :) You can find me on [LinkedIn](#) and [Twitter](#).

You can also check out my other blogposts [here](#).



Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



Get this newsletter

[About](#) [Help](#) [Terms](#) [Privacy](#)





Open in app

Get started

