Modules in Pytorch

Eunhui Kim/KISTI

https://fleuret.org/ee559/

Modules in Pytorch

목차

1) torch.nn.Module, torch.nn.functional

- torch.nn.functional 의 요소는 제공된 인수만으로 결과를 계산하는 autograd 호환 함수로, 보통 F로 import 됨. (e. g. import torch.nn.functional as F)

- torch.nn.Module 의 Subclass들은 losses 와 network components임.

- 후자(network components)는 훈련 중에 최적화 할 매개 변수를 포함함.

- 매개변수는 requires_grad 가 True인 torch.nn.Parameter 유형이며, 다양한 유틸리티 함수, 특히 torch.nn.Module.parameters(),의 모델 파라미터로 알려짐.

1) torch.nn.Module, torch.nn.functional

! torch.nn 함수와 모듈은 첫 번째 차원이 이를 인덱싱 하는 tensor에 저장된 입력 배치(batches of inputs)를 처리하고, 동일한 추가 차원을 갖는 해당 tensor를 생성함.

E.g. a fully connected layer $R^C \rightarrow R^D$ 는 크기 $N \times C$의 입력 tensor를 예상하고, $N \times D$크기의 tensor를 계산함. 여기서 N 은 샘플의 수이며 호출마다 다를 수 있음.

2) torch.nn.functional.relu

torch.nn.functional.relu(input, inplace=False)

모드 크기의 tensor를 입력으로 취할 수 있으며, 입력 tensor 각 값에 ReLU를 적용하여 동일한 크기의 결과 tensor를 생성함.

In [4]:

```
1 import torch
2 import torch.nn.functional as F
3
4 x = torch.tensor([[0.8008, -0.2586, 0.5019, -0.2002, -0.7416],
5                  [0.0557, 0.6046, 0.0864, -0.5929, 1.2606]])
6 print(x)
7
8 y = F.relu(x)
9 print(y)
```

```
tensor([[ 0.8008, -0.2586,  0.5019, -0.2002, -0.7416],
        [ 0.0557,  0.6046,  0.0864, -0.5929,  1.2606]])
tensor([[0.8008, 0.0000, 0.5019, 0.0000, 0.0000],
        [0.0557, 0.6046, 0.0864, 0.0000, 1.2606]])
```

inplace 는 연산 시 인수자체를 수정해야 하는지의 여부를 나타냄.
이는 처리 시 memory footprint를 줄이는 장점을 지님.

3) torch.nn.Linear

모듈

torch.nn.Linear(in_features, out_features, bias=True)

은 a $R^C \rightarrow R^D$ 완전 연결 계층을 구현함.
$N \times C$의 입력 tensor를 취해서 크기 $N \times D$의 출력 tensor를 생성함.

```
>>> f = nn.Linear(in_features = 10, out_features = 4)
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([4, 10])
bias torch.Size([4])
>>> x = torch.empty(523, 10).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([523, 4])
```

! 생성 시 weights와 biases는 자동으로 생성됨.
초기화 관련하여서 optimization에서 다시 다룰 예정임.

4) torch.nn.MSELoss()

모듈

torch.nn.MSELoss()

은 Mean Square Error loss를 구현함:
tensor의 **총 성분 수로 나눈** 성분 별 제곱 차이의 합

```
>>> f = torch.nn.MSELoss()
>>> x = torch.tensor([[ 3. ]])
>>> y = torch.tensor([[ 0. ]])
>>> f(x, y)
tensor(9.)
>>> x = torch.tensor([[ 3., 0., 0., 0. ]])
>>> y = torch.tensor([[ 0., 0., 0., 0. ]])
>>> f(x, y)
tensor(2.2500)
```

보통 loss의 첫 번째 매개변수는 input이고, 두 번째 매개변수는 target임.
target. 이 두 수량은 일부 손실에 대해 서로 다른 차원이거나 서로 다른 type일 수 있음. (e.g. for classification).

5) PyTorch- Batch Processing

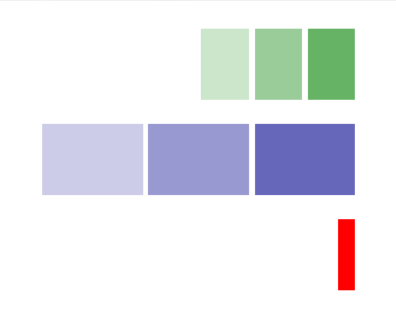
- Functions and modules from torch.nn 의 함수 및 모듈은 샘플들을 batches로 처리함. 이는 이를 유도하는 계산 속도 향상을 위함.
- 샘플의 모듈을 계산하기 위해서는 모듈의 매개변수와 샘플이 모두 cache memory로 먼저 복사되어야 함. 그런데 이 cache memory는 **빠르지만 작음**.
- 합리적인 크기의 모델에 대해 매개 변수의 일부만 cache에 유지가 가능함. 따라서 매개변수가 사용될 때 마다 모듈의 매개 변수가 cache에 복사되어야 함.
- 이러한 메모리 전달은 연산 그 자체보다 더 느릴 수 있음.
- 이것이 batch processing의 주된 이유이며,
  cache에 대한 매개 변수 사본 수를 batch당 module당 하나씩으로 줄임.
- 또한 매우 느린 Python loops 사용을 줄임.

5) PyTorch- Batch Processing

- 세 module로 구성된 model을 예시로 들어보면,

$$f = f_3 \circ f_2 \circ f_1$$

계산 되어야 하는 함수는 $f(x_1), f(x_2), f(x_3)$ 이 됨.

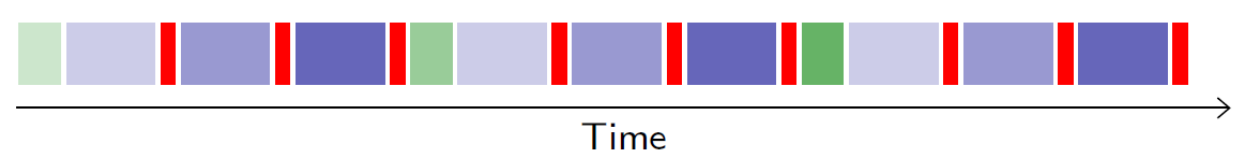


x_n을 cache memory로 복사

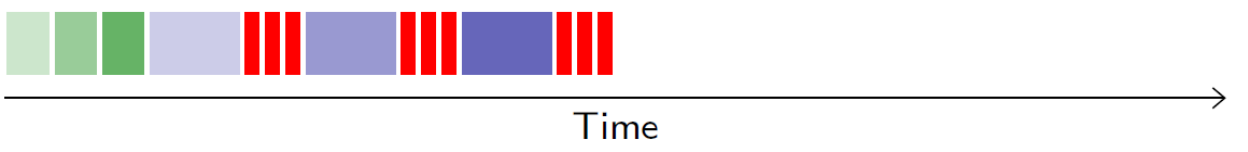
f_d 의 매개별수를 cache memory로 복사

$f_d(x_1)$ 를 계산

- 샘플들을 하나씩 처리하는 경우:



- Batch processing:



5) PyTorch- Batch Processing

```python
import time
def timing(x, w, batch = False, nb = 101):
    t = torch.zeros(nb)

    for u in range(0, t.size(0)):
        t0 = time.perf_counter()
        if batch:
            y = x.mm(w.t())
        else:
            y = torch.empty(x.size(0), w.size(0))
            for k in range(y.size(0)):
                y[k] = w.mv(x[k])
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0

    return t.median().item()

x = torch.empty(2500, 1000).normal_()
w = torch.empty(1500, 1000).normal_()
print('Batch-processing speed-up on CPU %.1f' %
      (timing(x, w, batch = False)/timing(x, w, batch = True)))

x = x.to('cuda')
w = w.to('cuda')
print('Batch-processing speed-up on GPU %.1f' %
      (timing(x, w, batch = False)/timing(x, w, batch = True)))


```

```
Batch-processing speed-up on CPU 16.4
Batch-processing speed-up on GPU 102.8
```

5) PyTorch- Batch Processing

- 공식적으로 이전에 살펴본 완전연결계층(fully connected layer)에 대해 몇 가지 표현을 다시 검토해 보면,

$$\forall l, n, w^{(l)} \in R^{d_l \times d_{l-1}}, x_n^{(l-1)} \in R^{d_{l-1}}, s_n^{(l)} = w^{(l)} \times x_n^{(l-1)}$$

- 이제부터는 행 벡터를 사용하여 일련의 샘플을 첫 번째 인덱스가 샘플의 인덱스인 2D 배열로 나타낼 수 있음.

$$x = \begin{pmatrix} x_{1,1} & \cdots & x_{1,D} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \cdots & x_{N,D} \end{pmatrix} = \begin{pmatrix} (x_1)^T \\ \vdots \\ (x_N)^T \end{pmatrix},$$

이는 $R^{N \times D}$ 의 요소임.

5) PyTorch– Batch Processing

- 모든 샘플 행 벡터를 만들고 선형 연산자(linear operator)를 적용하려면,

$$\forall\, n, s_n^{(l)} = \left(w^{(l)}\left(x_n^{(l-1)}\right)^T\right)^T = x_n^{(l-1)}\left(w^{(l)}\right)^T$$

- 전체 batch에 대해 tensorial 표현을 제공함

$$s^{(l)} = x_n^{(l-1)}\left(w^{(l)}\right)^T$$

- And in torch/nn/functional.py

In [16]:

```
def linear(input, weight, bias=None):
    if input.dim() == 2 and bias is not None:
        #fused op is marginally faster
        return torch.addmm(bias, input, weight.t())

    output = input.matmul(weight.t())
    if bias is not None:
        output += bias
    return output

```

5) PyTorch- Batch Processing

- 유사하게 선형 계층의 backward pass는다음과 같이 표현됨

$$\left[\frac{\partial L}{\partial w^{(l)}}\right] = \left[\frac{\partial L}{\partial x^{(l)}}\right]^{T} x^{(l-1)}$$

그리고

$$\left[\frac{\partial L}{\partial x^{(l)}}\right] = \left[\frac{\partial L}{\partial x^{(l+1)}}\right]^{T} w^{(l+1)}$$

6) Pytorch – Convolutionals

- 정상적인 "구조화되지 않은"벡터로 취급되는 경우 사운드 샘플 또는 이미지와 같은 대형 신호에는 다루기 힘든 크기의 모델이 필요하게 됨.

- 예를 들어 256x256 RGB 이미지를 입력으로 사용하고 동일한 크기의 이미지를 생성하는 선형 레이어에는

$$(256 \times 256 \times 3)^2 \approx 3.87e + 10$$

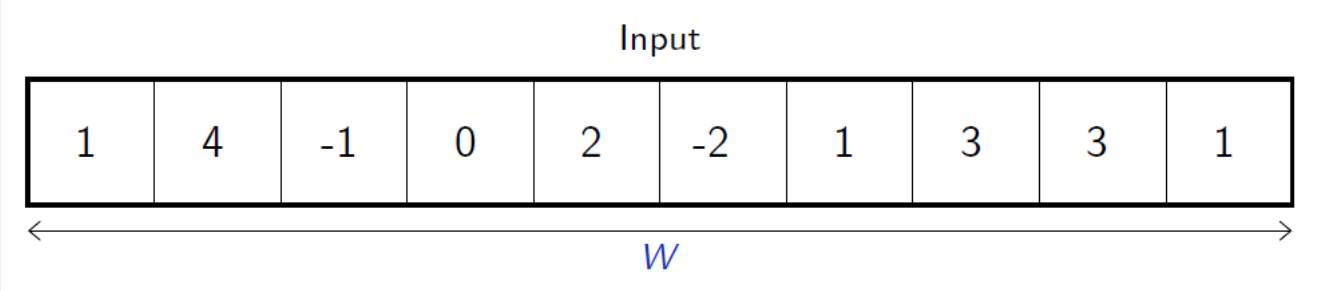
해당 메모리 foot print(≈ 150Gb!) 및 초과 용량과 함께 매개 변수가 필요함.

6) Pytorch – Convolutionals

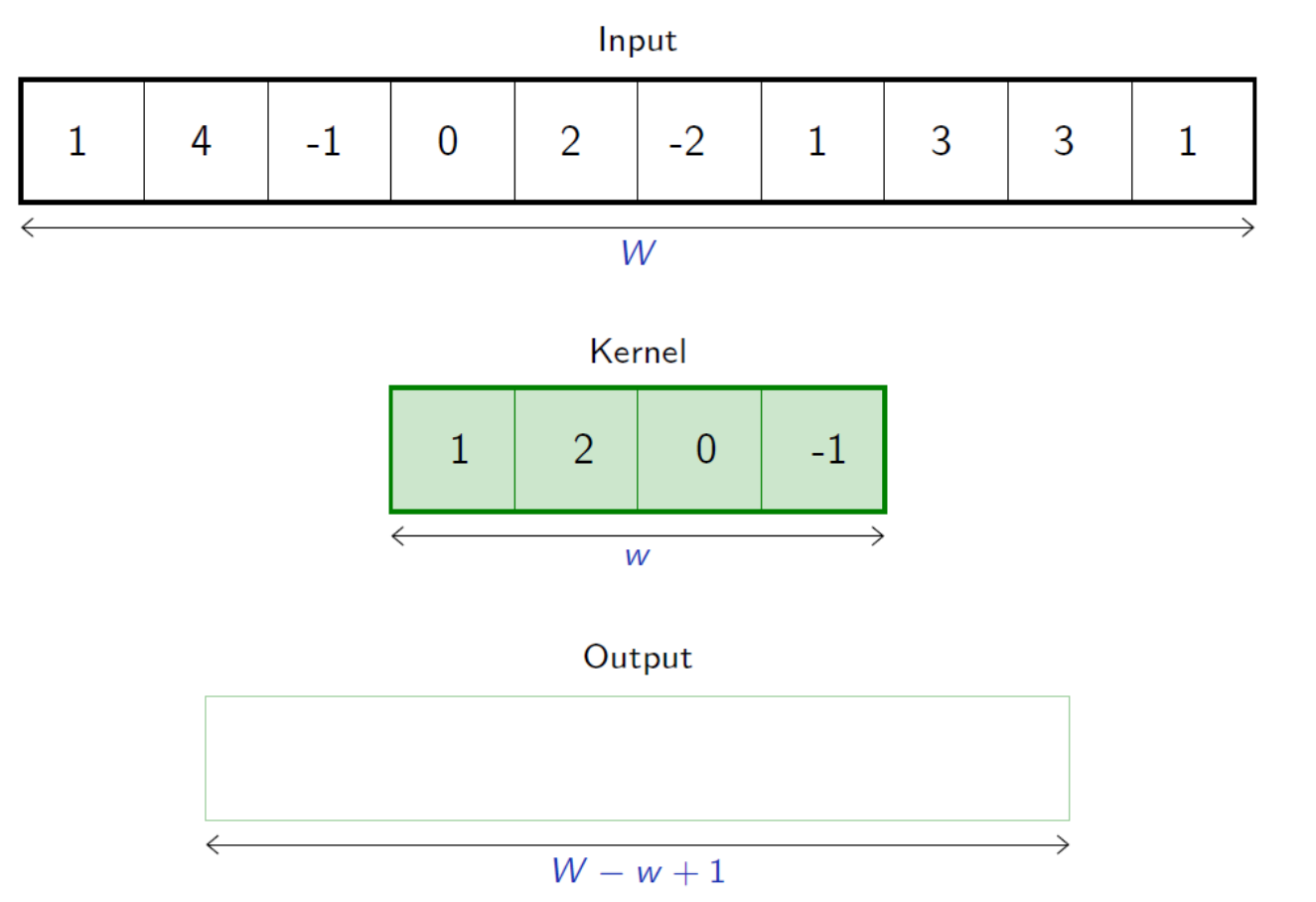
- 더욱이,이 요구 사항은 그러한 큰 신호가 " invariance in translation "을 갖는다는 직관과 일치하지 않음. **특정 위치에서 의미 있는 표현은 모든 곳에서 사용될 수 있어야 함.**

- A convolution layer는 이 아이디어를 구현한 것임. 즉, 동일한 선형 변환을 로컬, 모든 곳에 적용하고 신호 구조를 보존함.

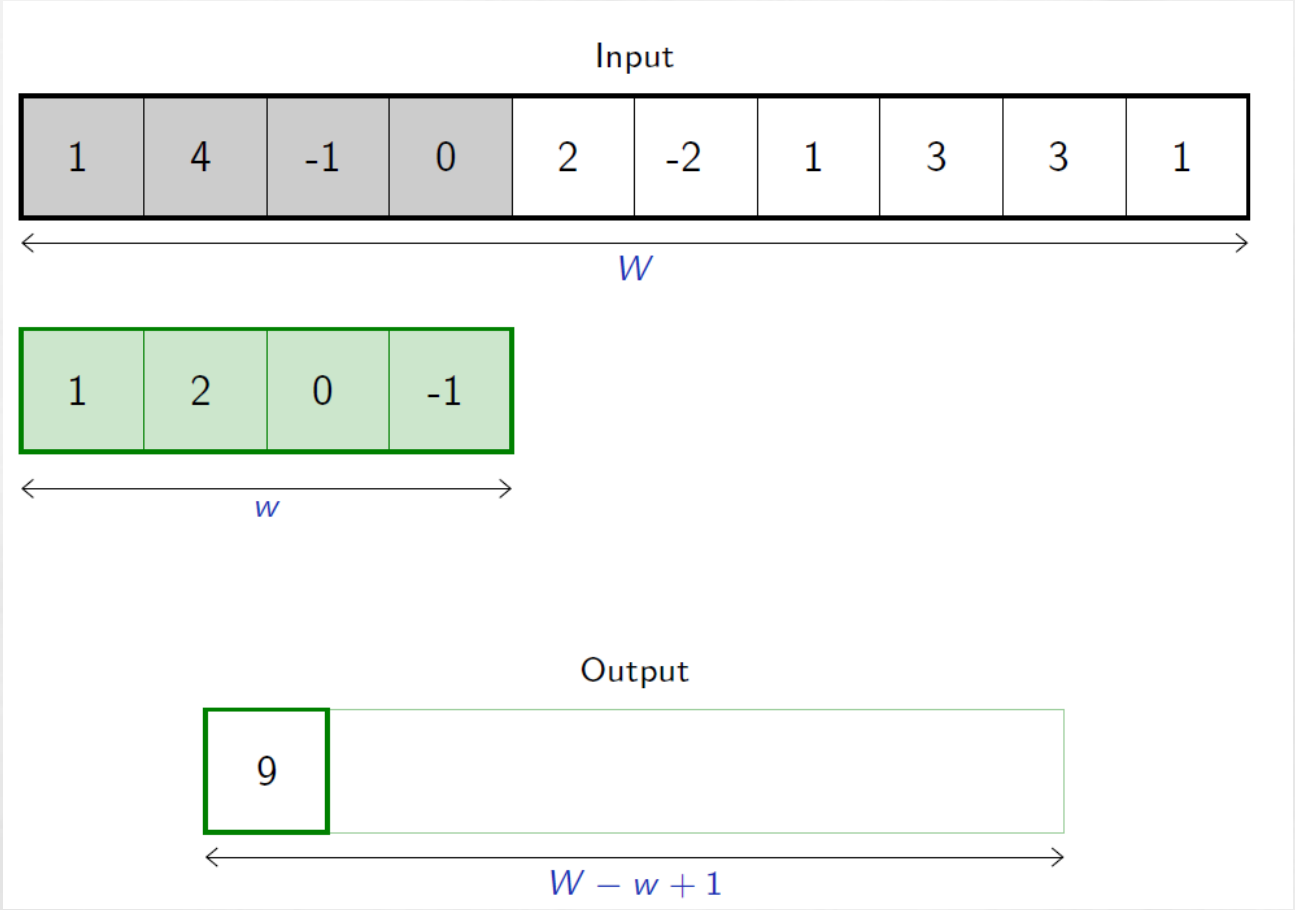
6) Pytorch – Convolutionals



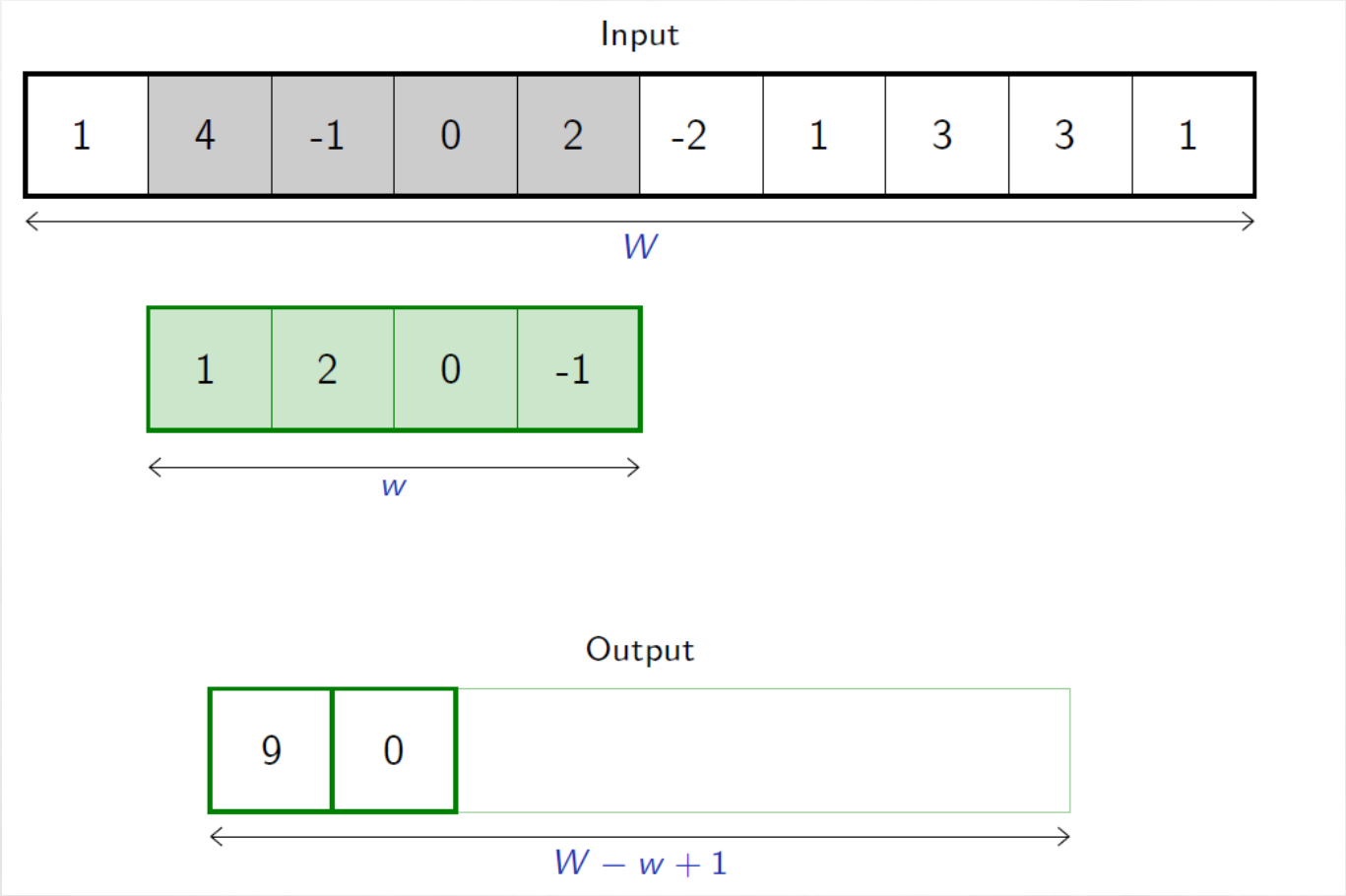
6) Pytorch – Convolutionals



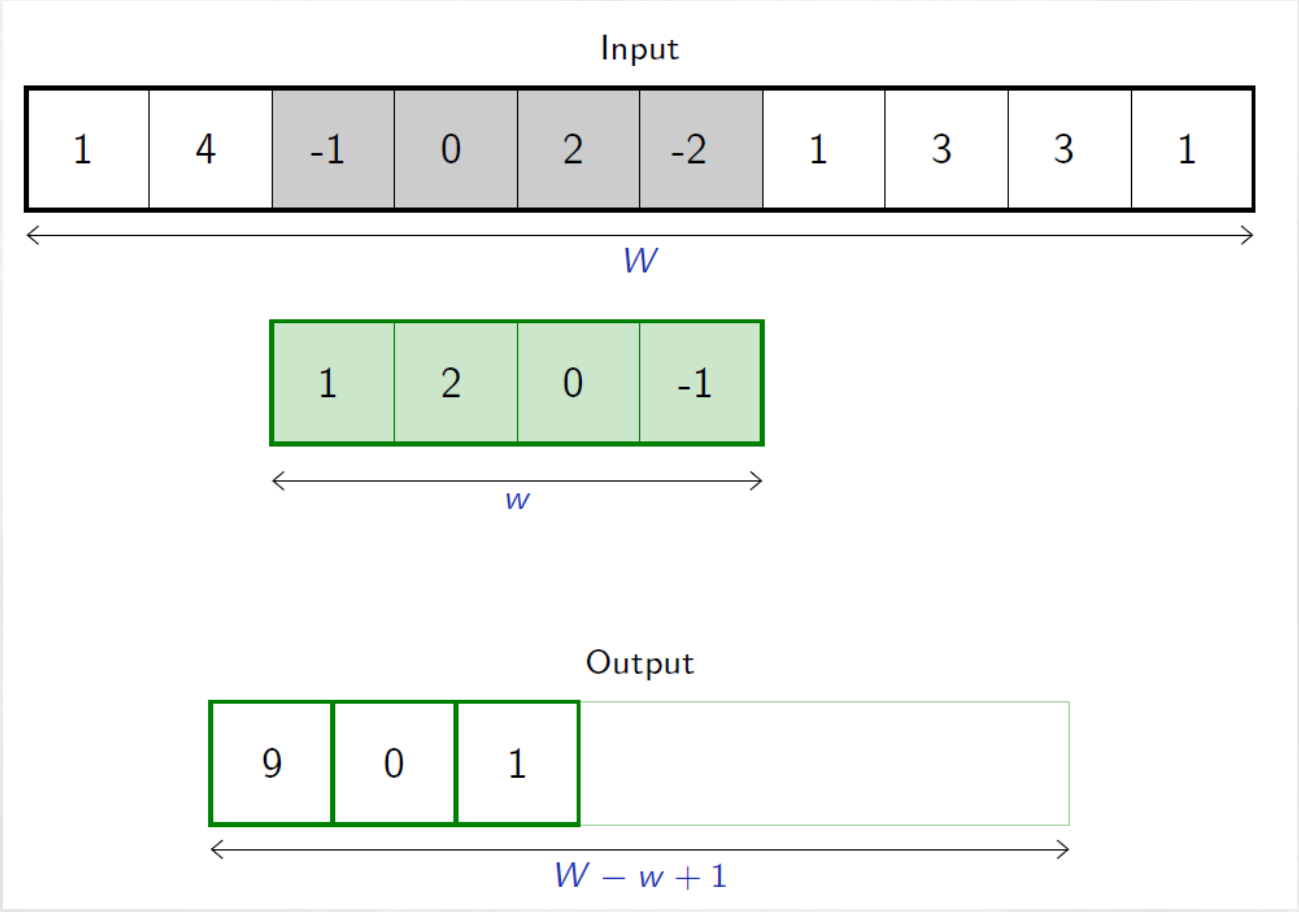
6) Pytorch – Convolutionals



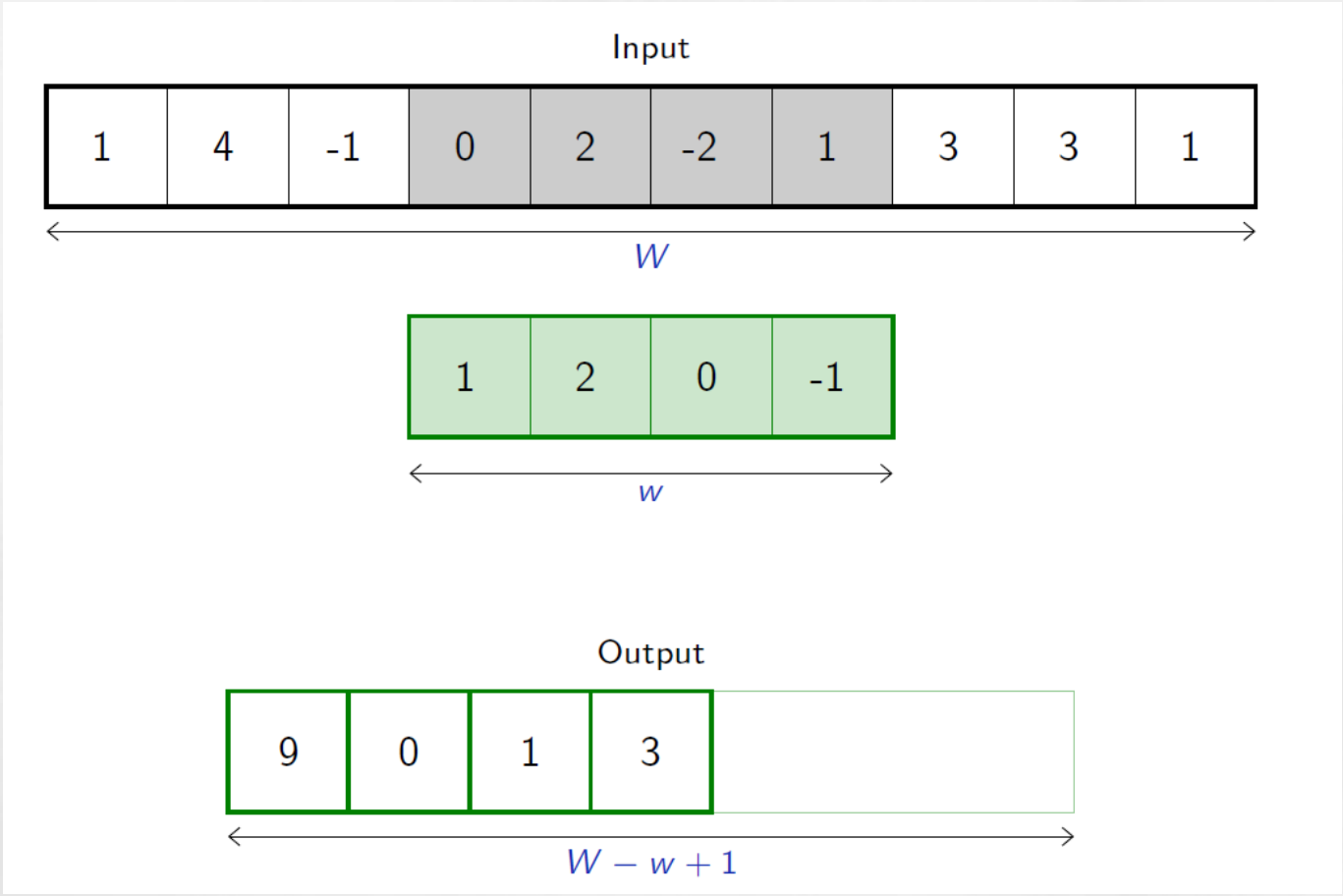
6) Pytorch – Convolutionals



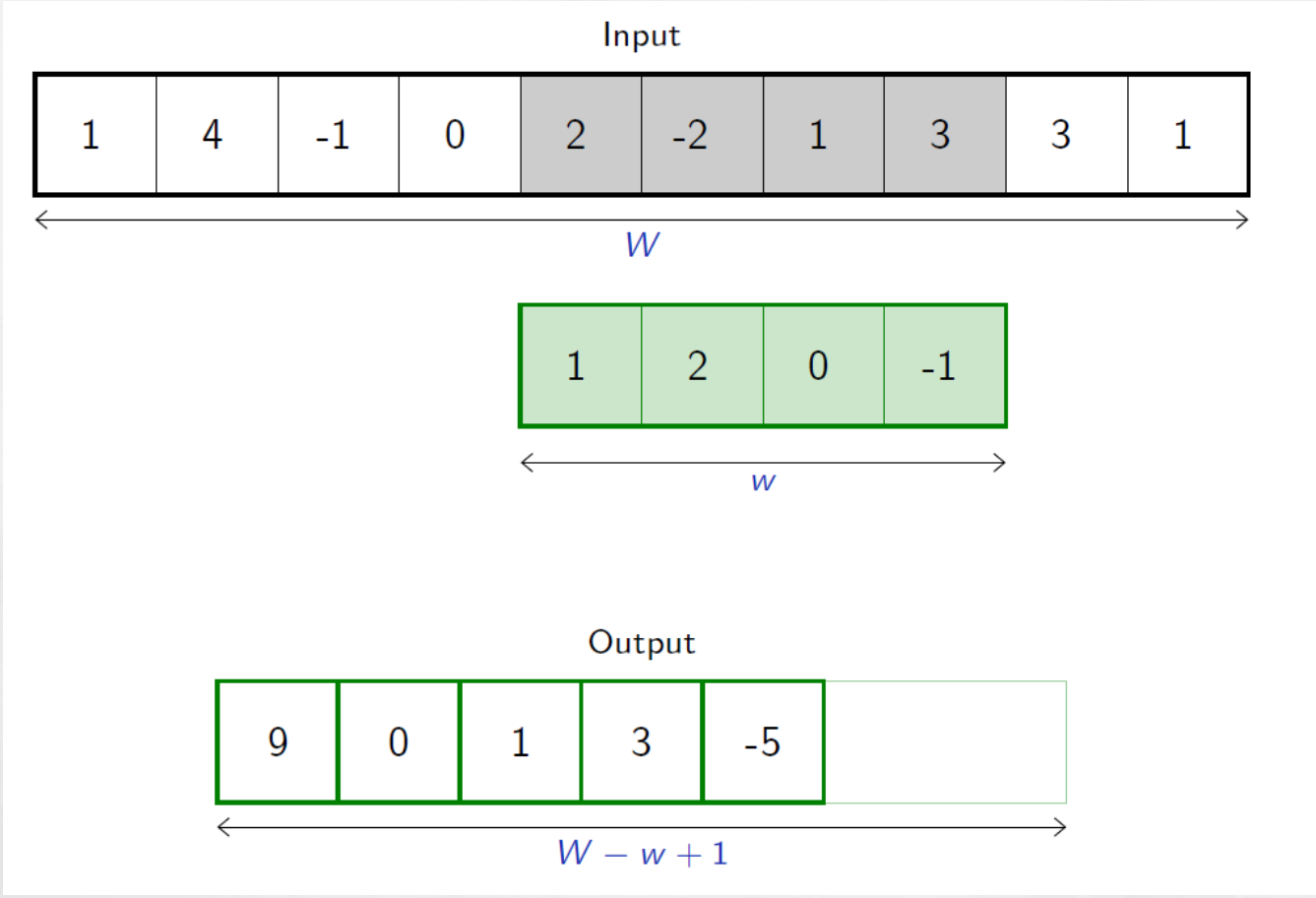
6) Pytorch – Convolutionals



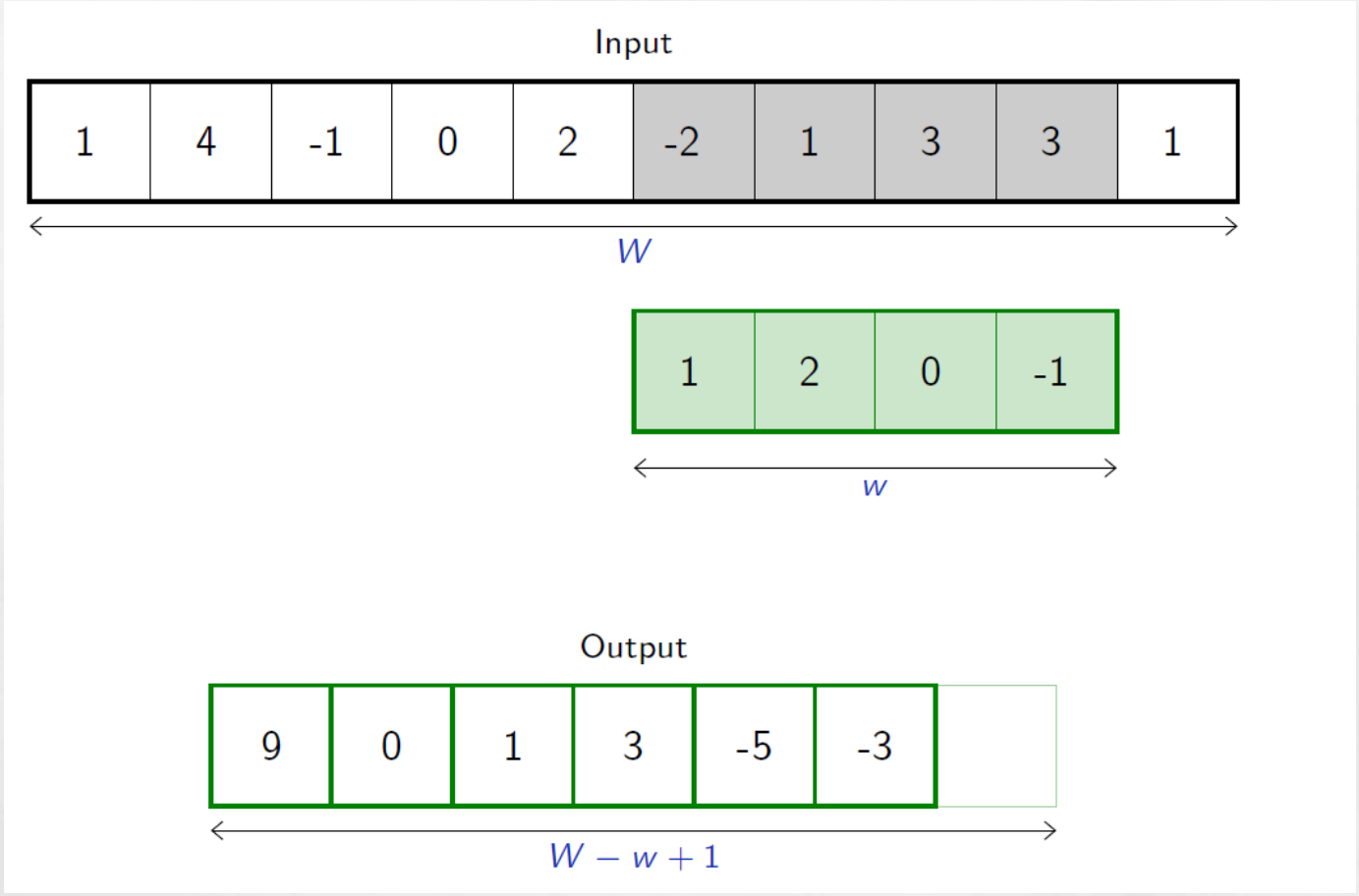
6) Pytorch – Convolutionals



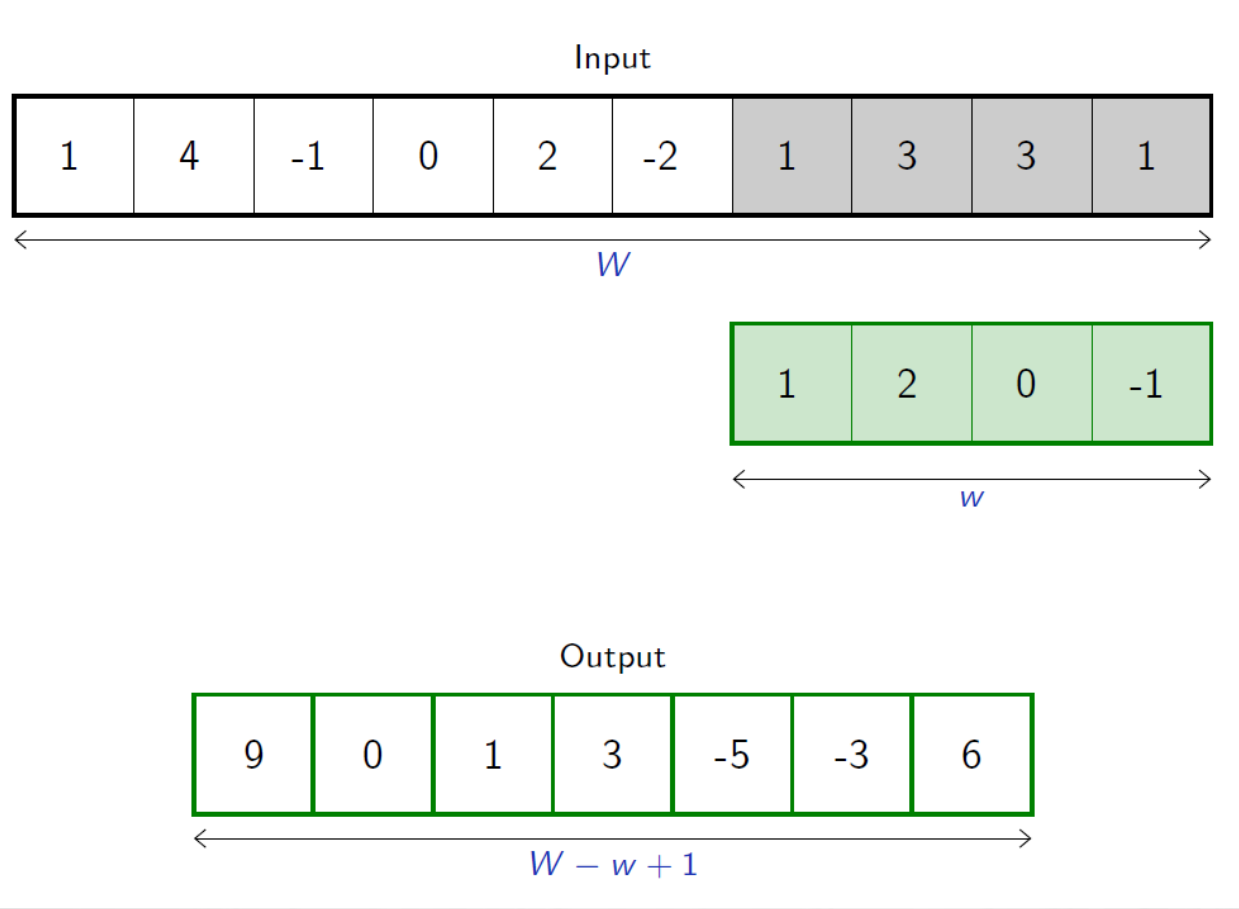
6) Pytorch – Convolutionals



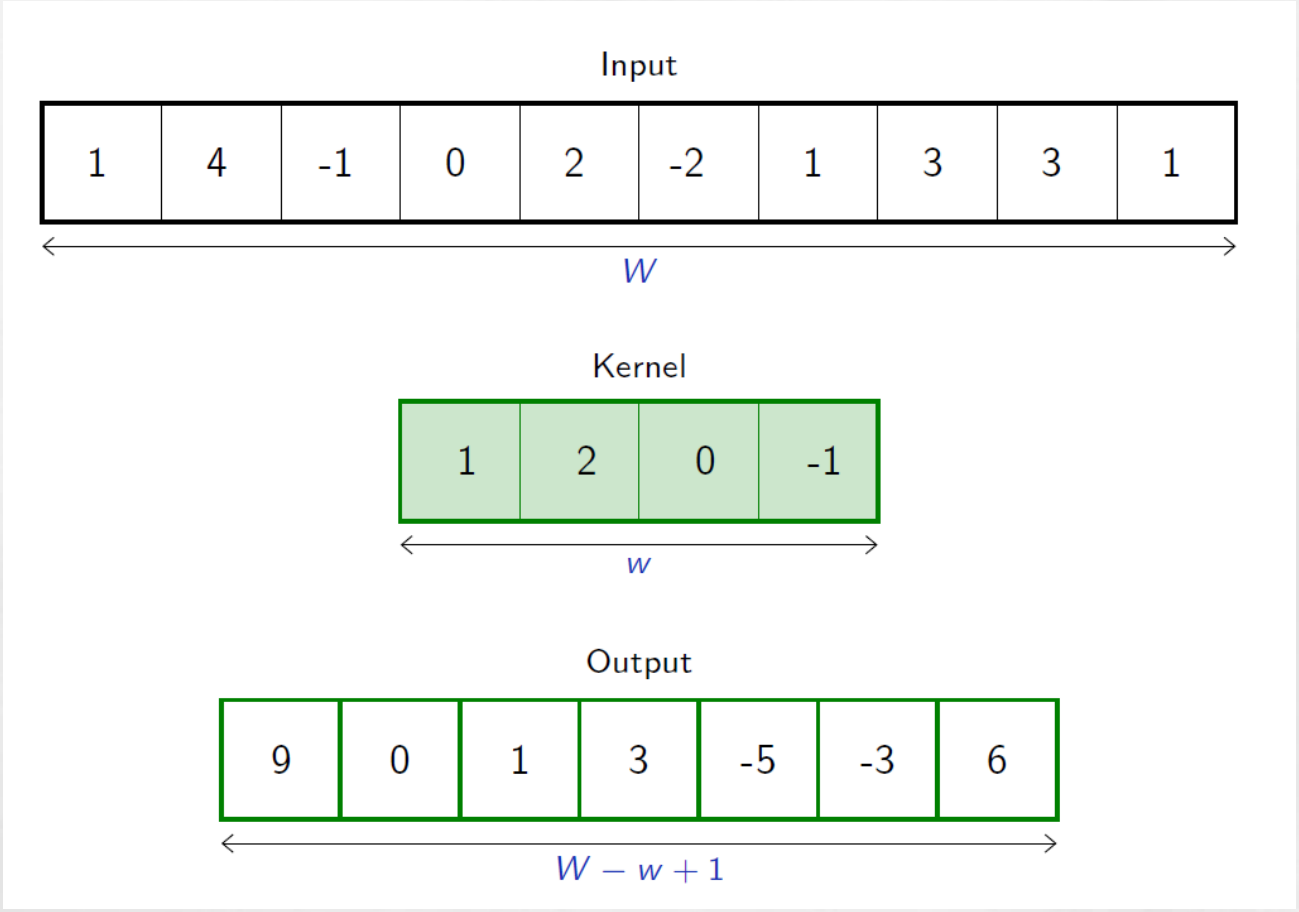
6) Pytorch – Convolutionals



6) Pytorch – Convolutionals



6) Pytorch – Convolutionals



6) Pytorch – Convolutionals

- 공식적으로, 1d에서, 입력 값이 다음과 같이 주어질 때,

$$x = (x_1, \cdots, x_W)$$

- 그리고 너비 w의 "convolution kernel" (혹은 "filter") 가 주어진 경우,

$$u = (u_1, \cdots, u_W)$$

- convolution 연산 $x * u$ 은 크기 $W - w + 1$ vector이다.

$$(x \circledast u)_i = \sum_{j=1}^{w} x_{i-1+j}\, u_j$$
$$= (x_i, \ldots, x_{i+w-1}) \cdot u$$

예를 들어,

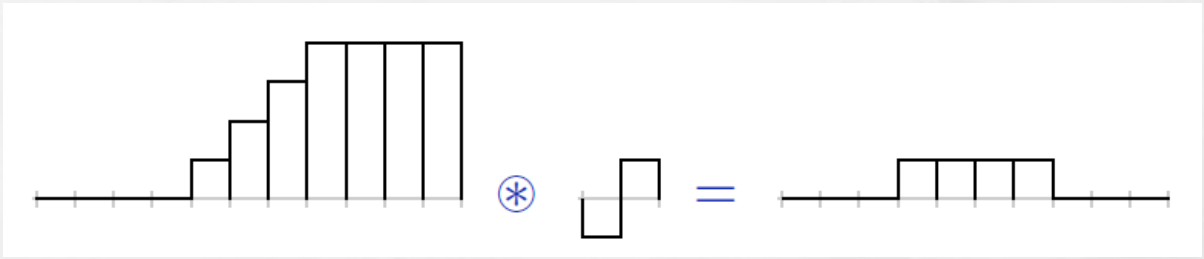
$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).$$

! 이것은 커널과 신호가 모두 인덱스 순서로 증가하기 때문에 일반적인 convolution과 다름.

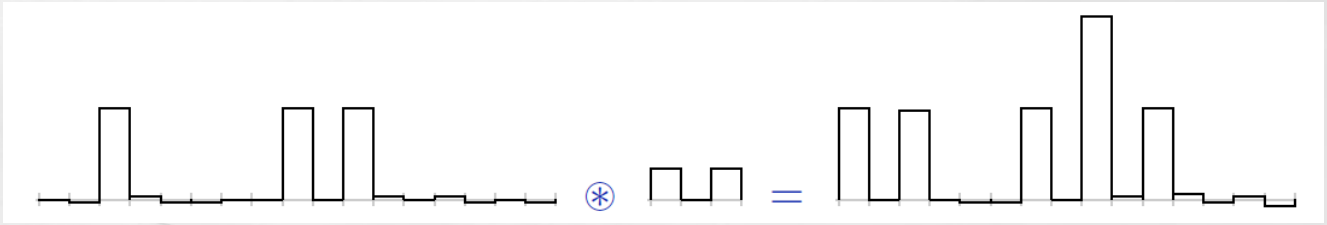
6) Pytorch – Convolutionals

- Convolution은 특히 미분 연산자를 구현할 수 있음. e.g.

$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



- 혹은 “template matcher", e.g.



- 두 연산 예시 모두 실제로 “invariant by translation ” 하다.

6) Pytorch – Convolutionals

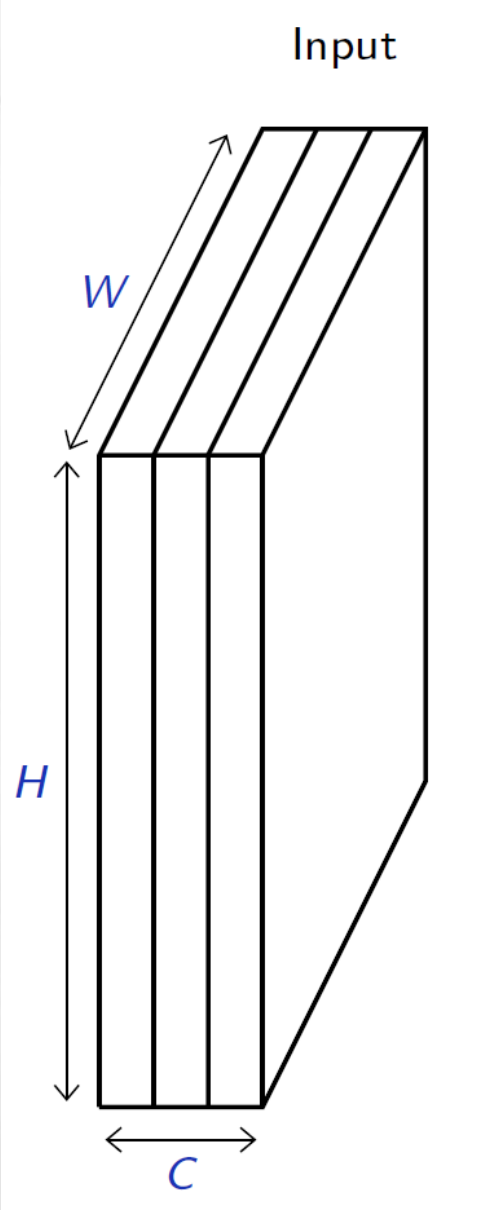
- 특정 application에 따라 복잡해질 수 있지만,
  일반적으로 다차원 입력으로 일반화 될 수 있음.

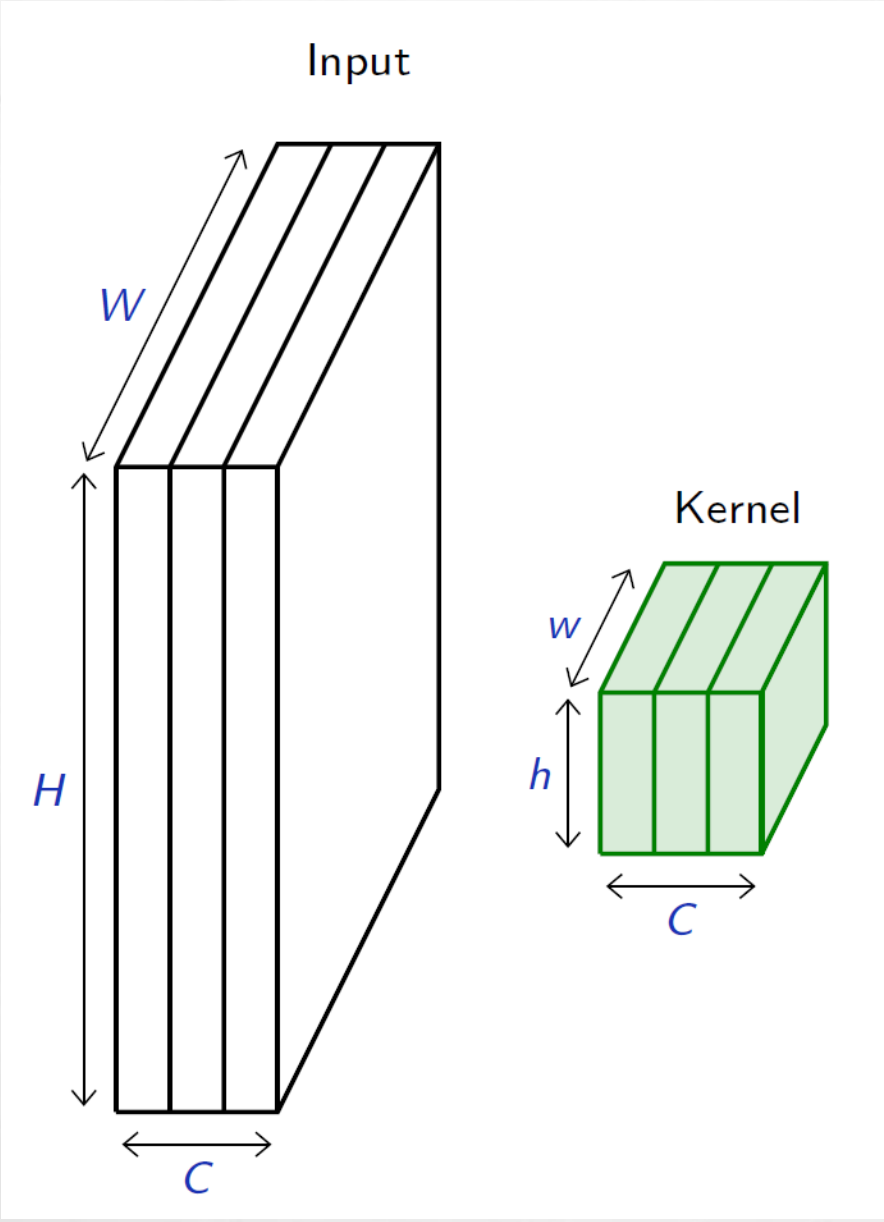
- “convolutional networks" 의 가장 일반적인 형태는
3D 텐서를 입력 (즉, 다중 채널 2D 신호)으로 처리하여 2D 텐서를 출력함.
kernel은 행과 열을 가로 질러 채널간에 스와이프되지 않음.

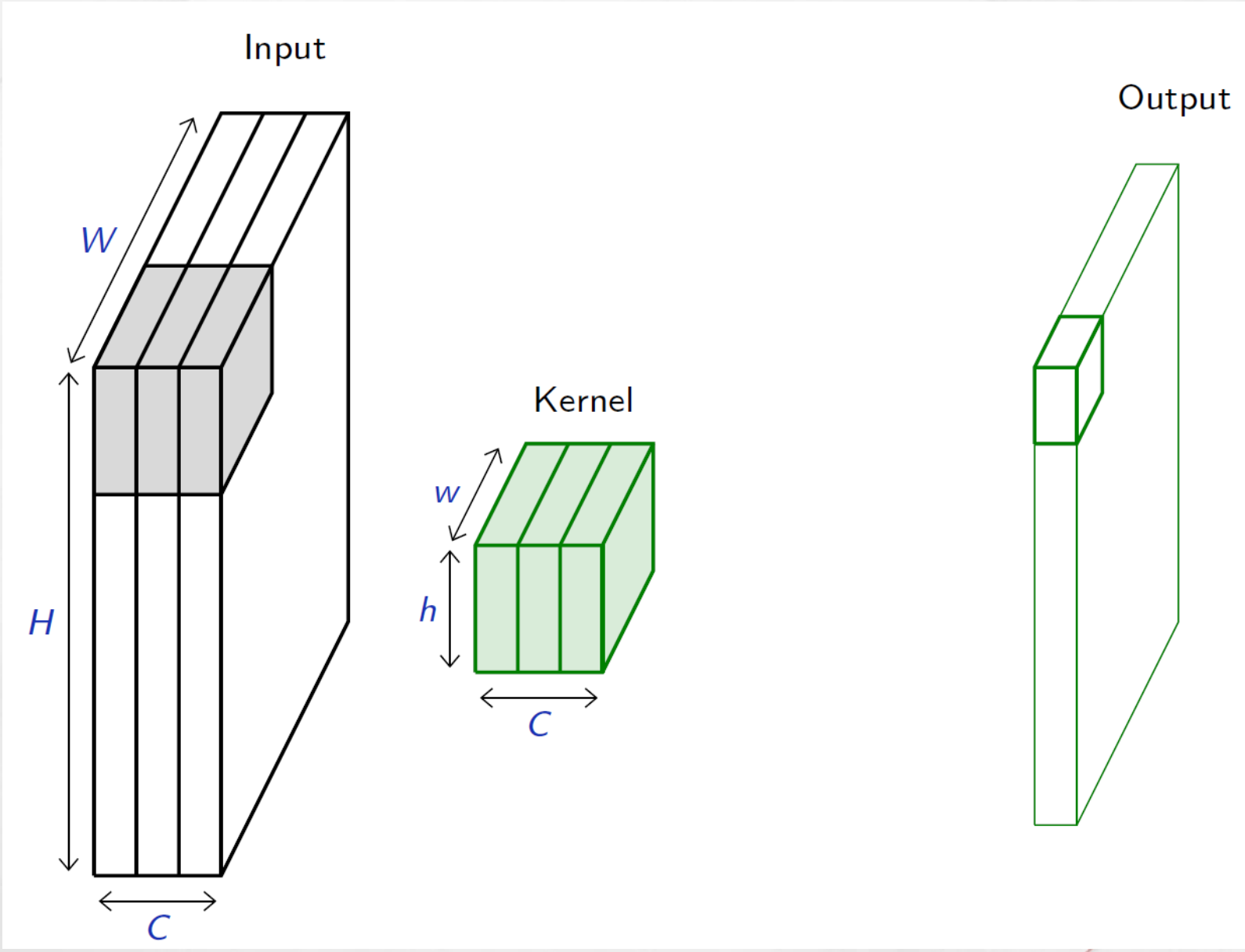
- 이러한 경우, 만일 입력 tensor가 크기 $C \times H \times W$이고 kernel이
$C \times h \times w$ 인 경우, 출력은 크기 $(H - h + 1) \times (W - w + 1)$임.

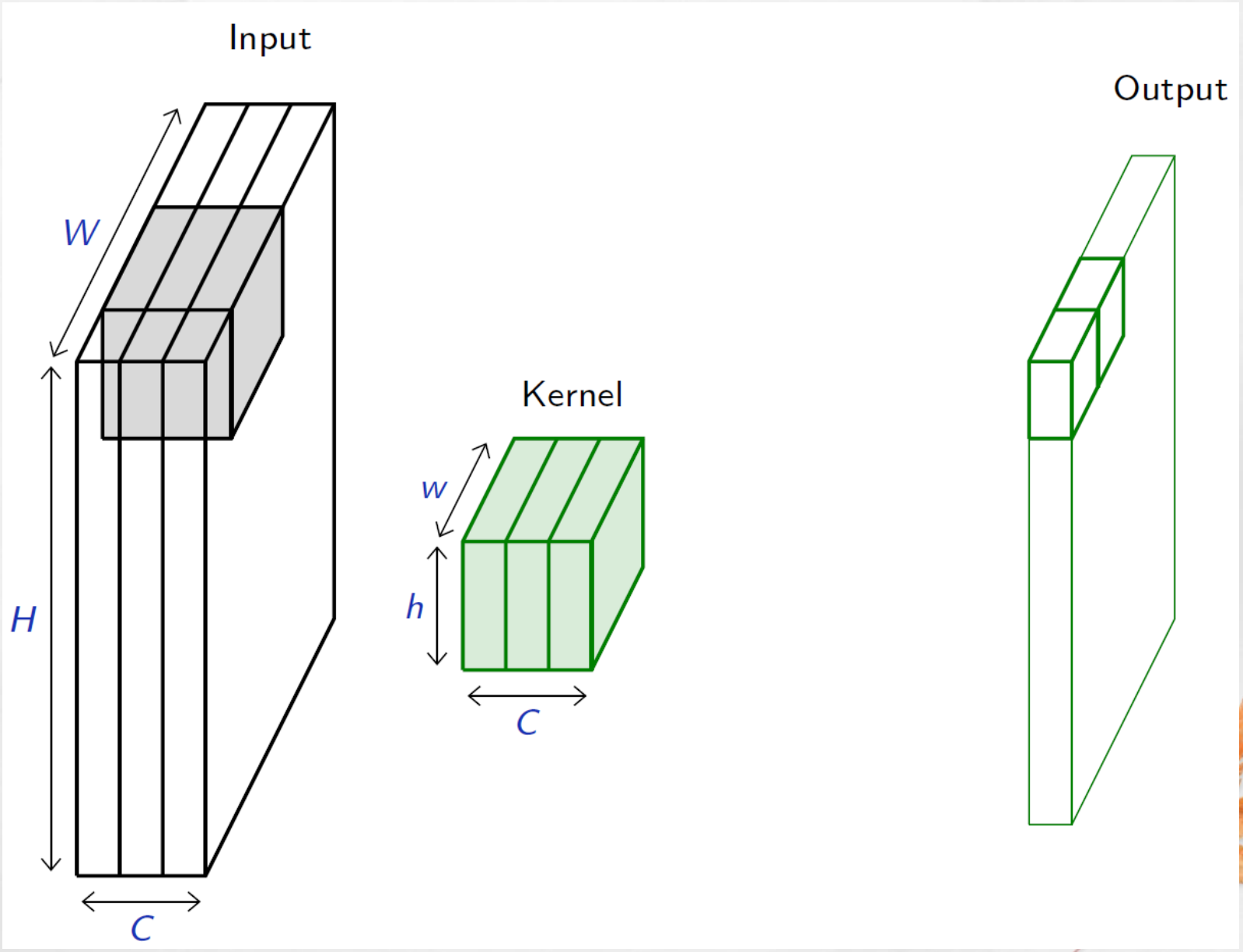
! C 채널이 있더라도 “2d 신호라" 함, feature index에 구조가 없는
2D 위치에 의해 index되는 feature vector이기 때문임.

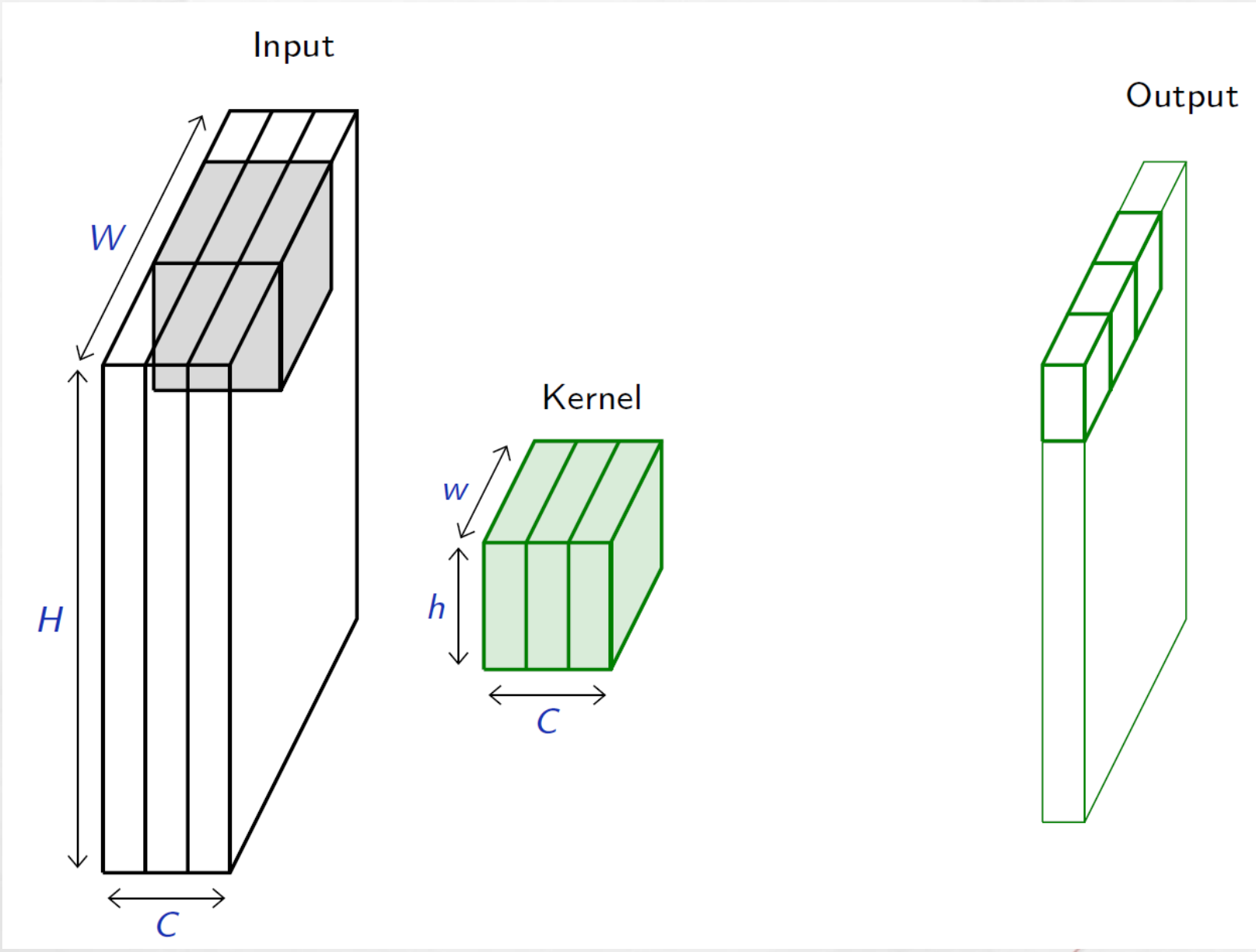
- 표준 convolution 계층에서 , D 개의 convolutions은 결합되어
$D \times (H - h + 1) \times (W - w + 1)$ 개의 출력을 생성함.

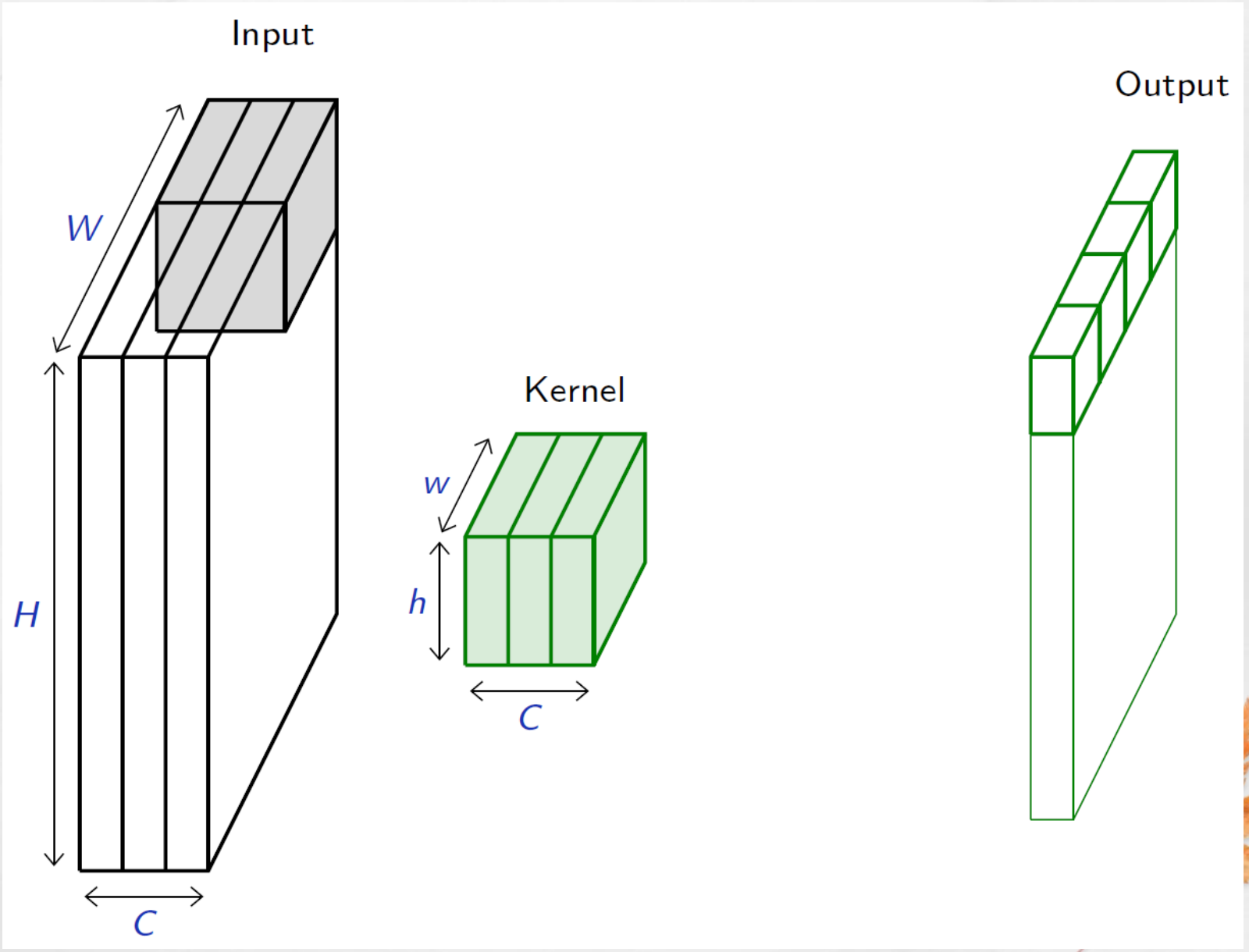

Input
W
H
C

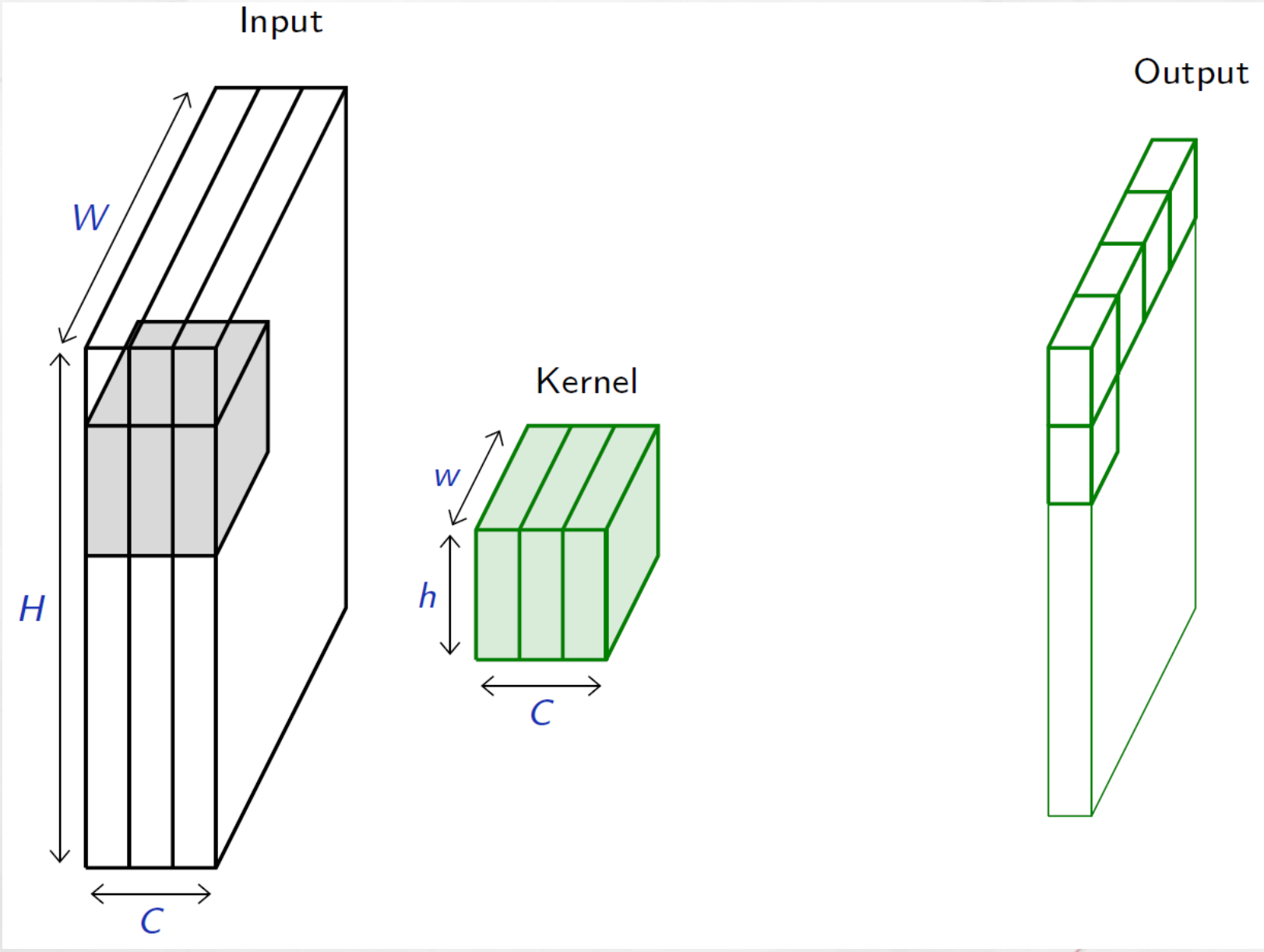

Input
W
H
C
Kernel
w
h
C

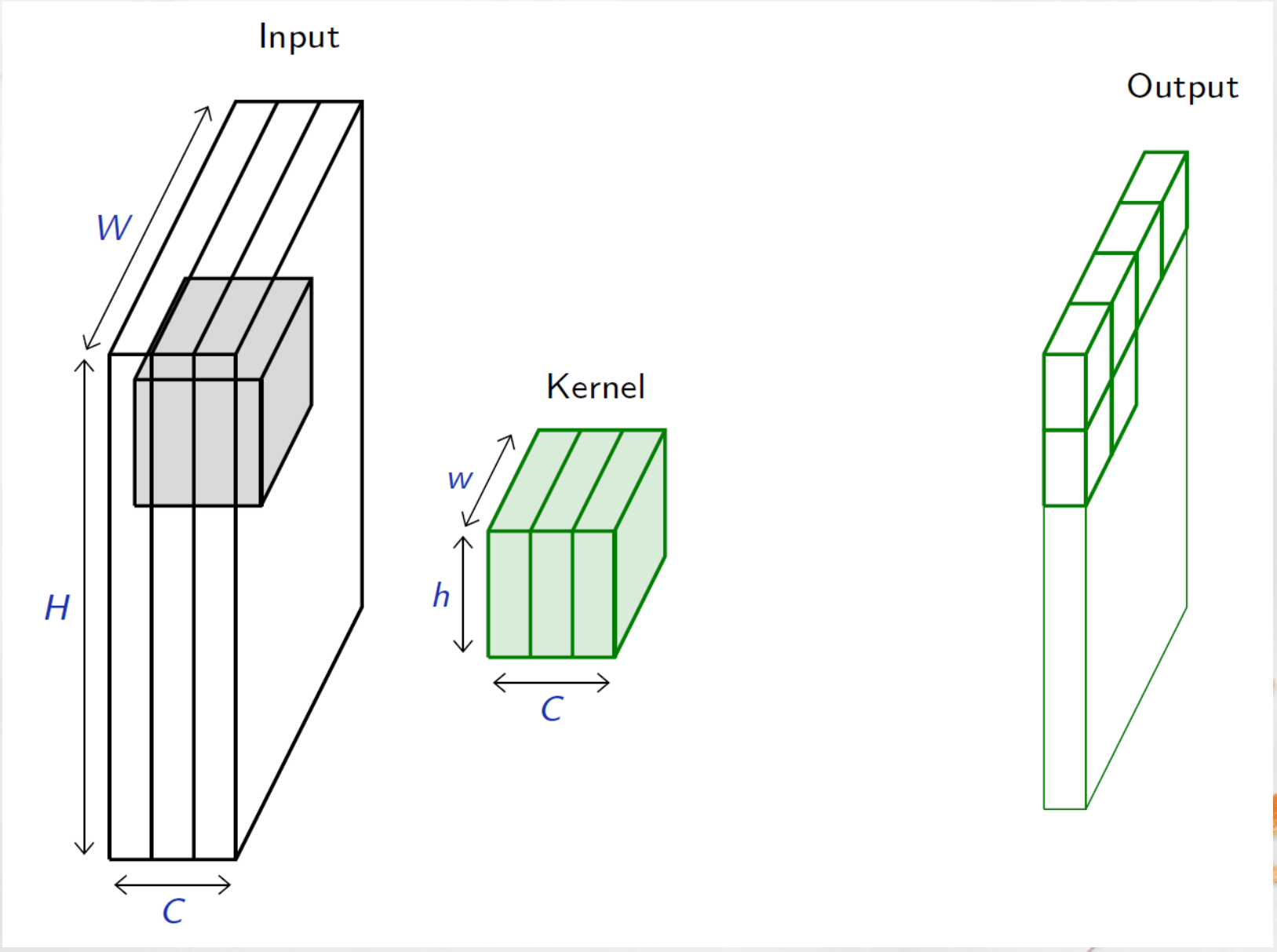

Input
W
H
C
Kernel
w
h
C
Output

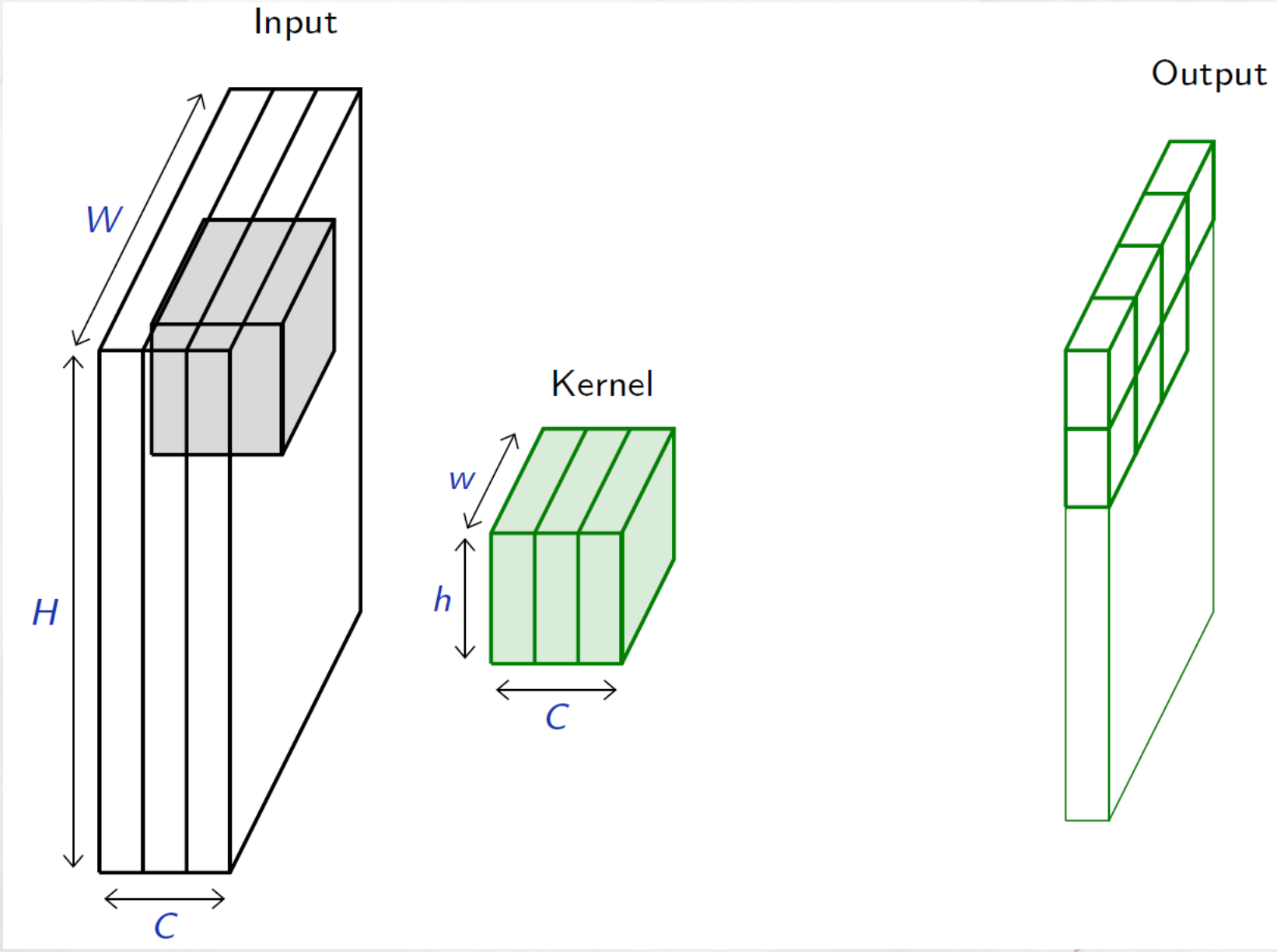

Input
W
H
C
Kernel
w
h
C
Output

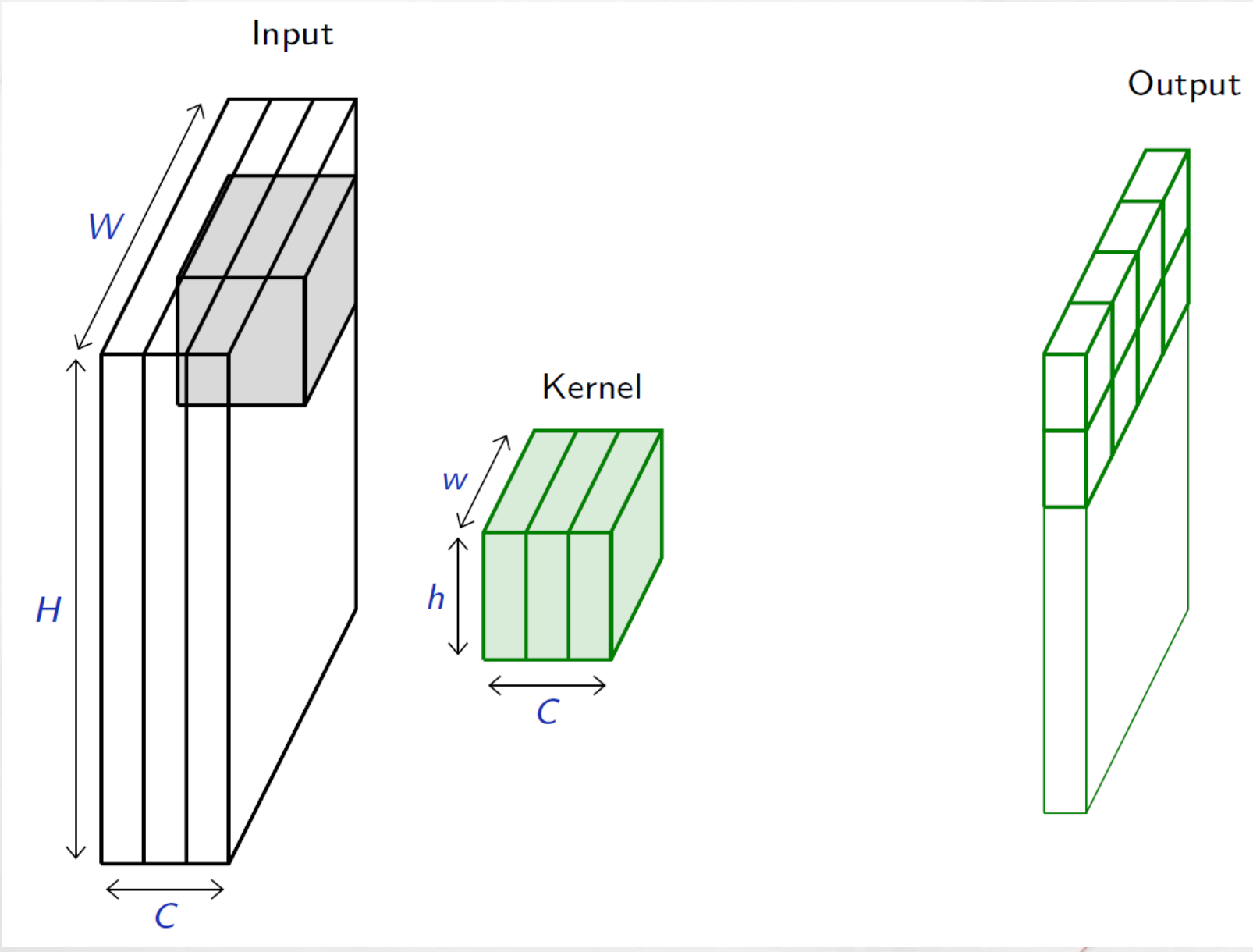

Input
W
H
C
Kernel
w
h
C
Output

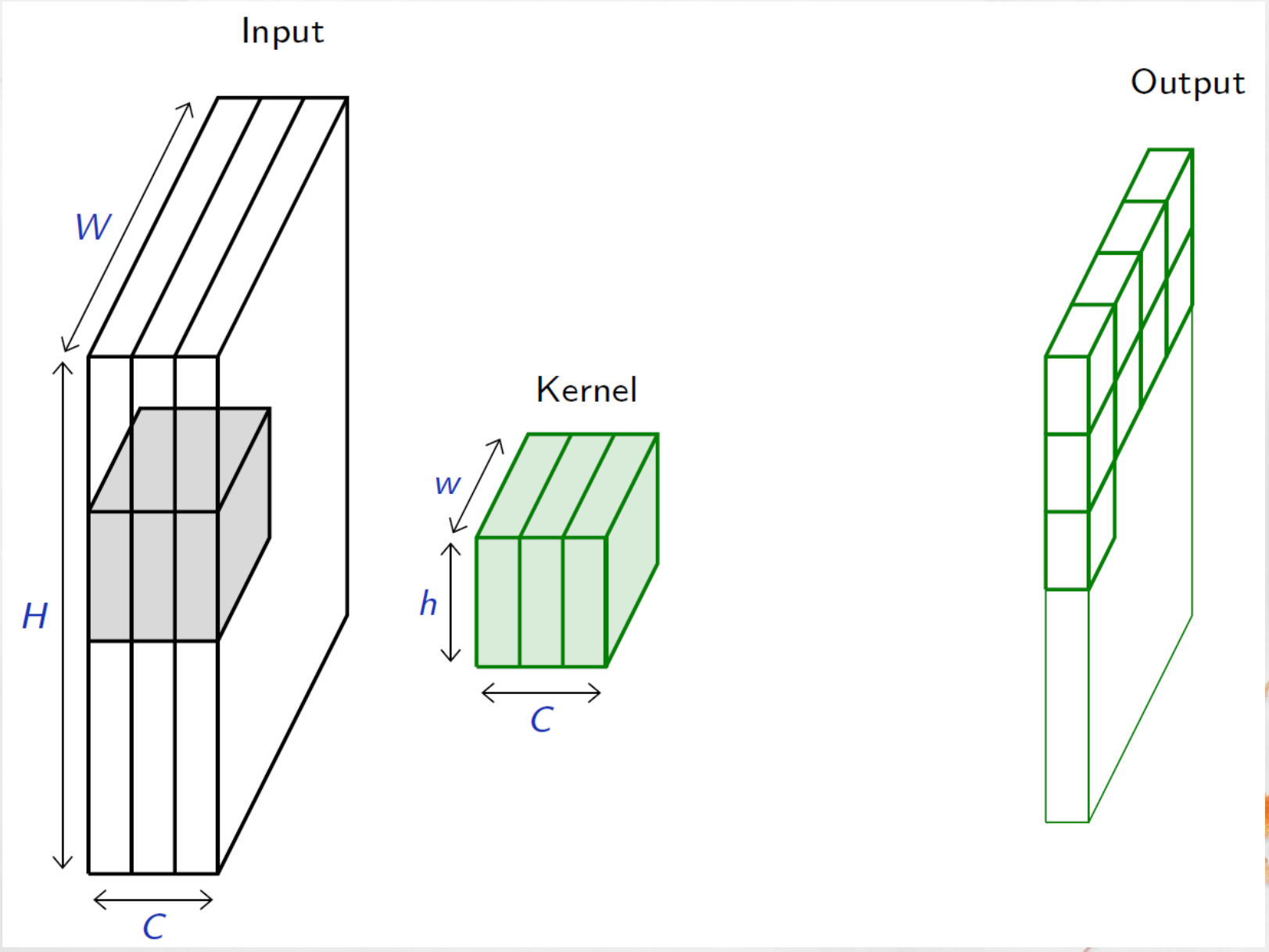

Input
W
H
C
Kernel
w
h
C
Output

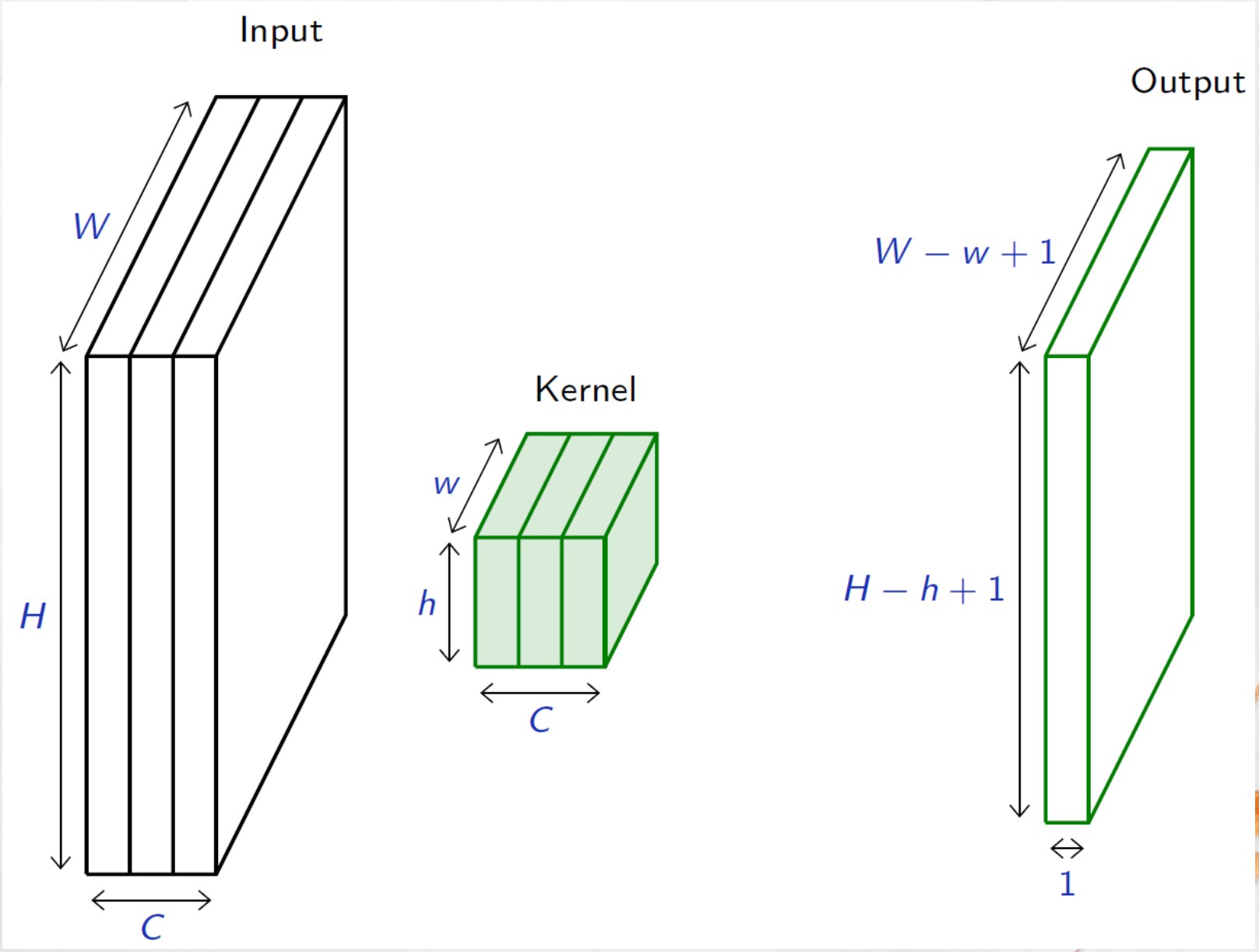

Input
W
H
C
Kernel
w
h
C
Output

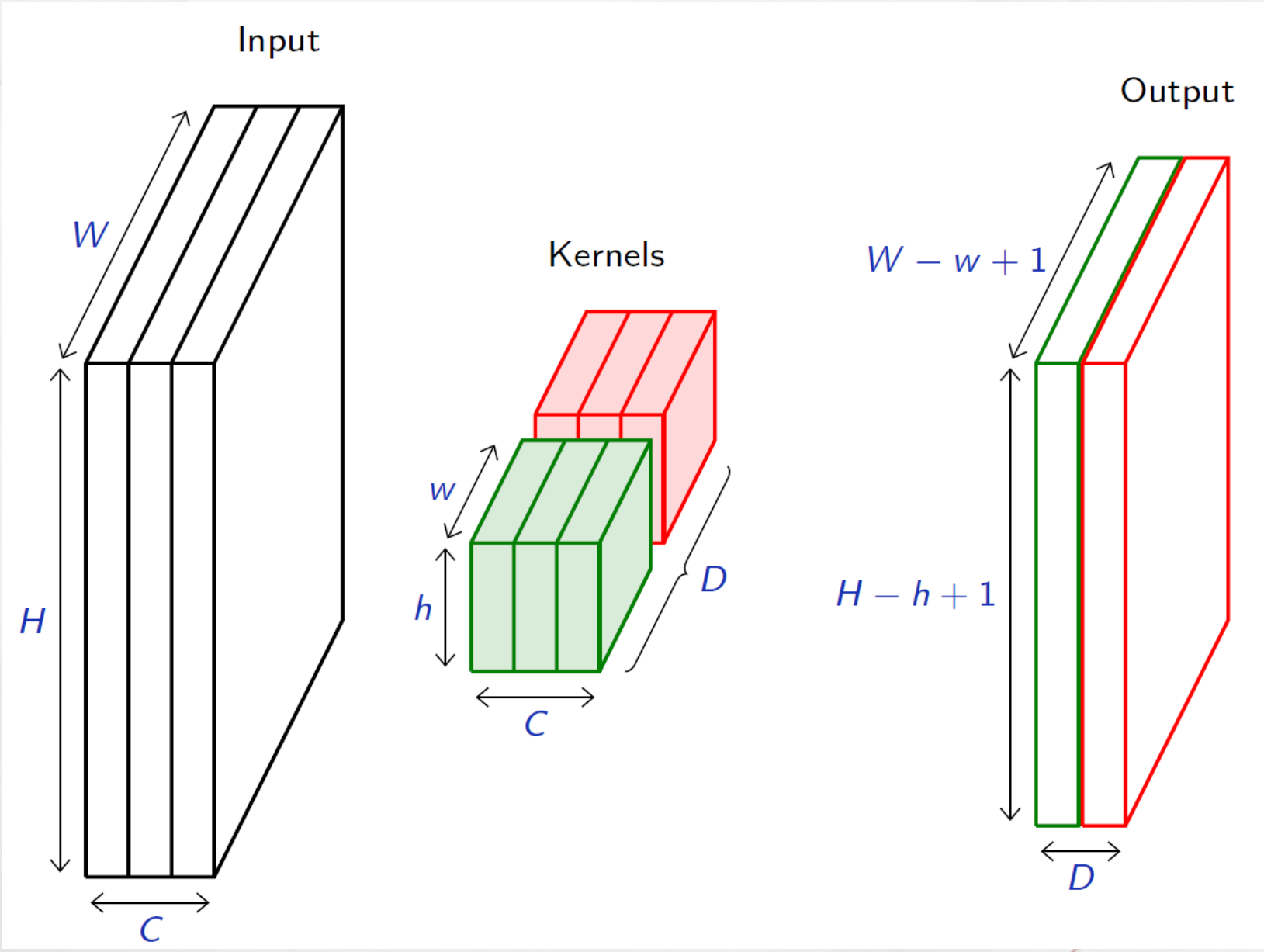

Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output
W − w + 1
H − h + 1
1


Input
W
H
C
Kernels
w
h
D
C
Output
W − w + 1
H − h + 1
D

6) Pytorch – Convolutionals

- convolution은 신호 지원 구조를 유지함.

- 1d 신호는 1d 신호로 전환되고, 2d신호는 2d신호로 전환됨.

  입력 신호의 이웃한 부분들은 출력 신호의 이웃한 부분들에 영향을 미치게 됨.

- 채널 인덱스가 일련의 회색조 비디오 프레임에서 시간과 같은 metric 의미인 경우, 3d convolution 을 사용할 수 있음. 그런 것이 아니라면,
채널을 스 와이프하는 것은 의미가 없음.

6) Pytorch – Convolutionals

- 보통 convolution layer에서 생성한 채널 중 하나를 activation map으로 참조함
- 입력의 부분 영역들이 activation map의 receptive field로서 출력 요소에 영향을 미침.
- convolutional networks의 context에서 표준 linear계층은 완전연결계층(a fully connected layer)이라 부르는데 입력이 모두 출력에 직접 영향을 미치기 때문임.

6) Pytorch – Convolutionals

torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)

- 가중치가 크기 $D \times C \times h \times w$인 커널을 포함하고, 크기 D 인 bias를 포함하는 2d convolution의 입력 차원은 $N \times C \times H \times W$

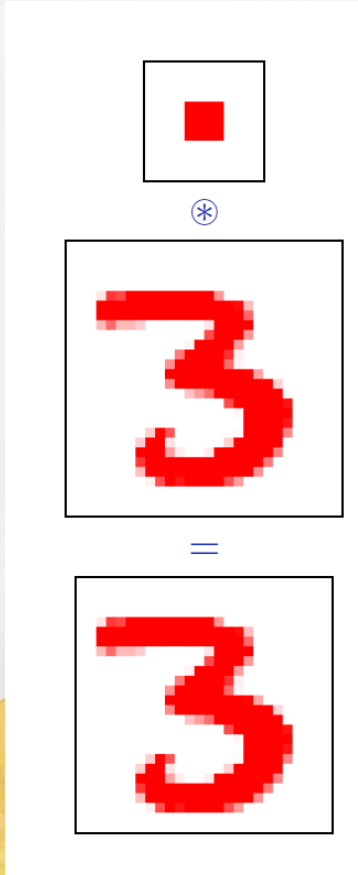
  이며 결과 차원은 $N \times D \times (H - h + 1) \times (W - w + 1)$ 임.

In [17]:
```
1 weight = torch.empty(5, 4, 2, 3).normal_()
2 bias = torch.empty(5).normal_()
3 input = torch.empty(117, 4, 10, 3).normal_()
4 output = torch.nn.functional.conv2d(input, weight, bias)
5 output.size()
```

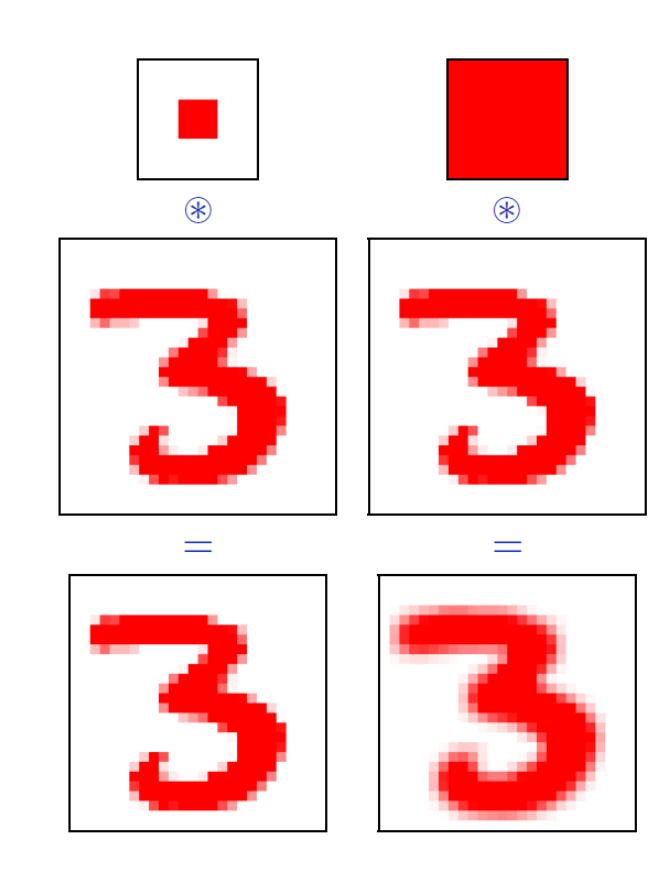
Out[17]:
```
torch.Size([117, 5, 9, 1])
```

- 유사하게 1d 와 3d convolutions을 구현함

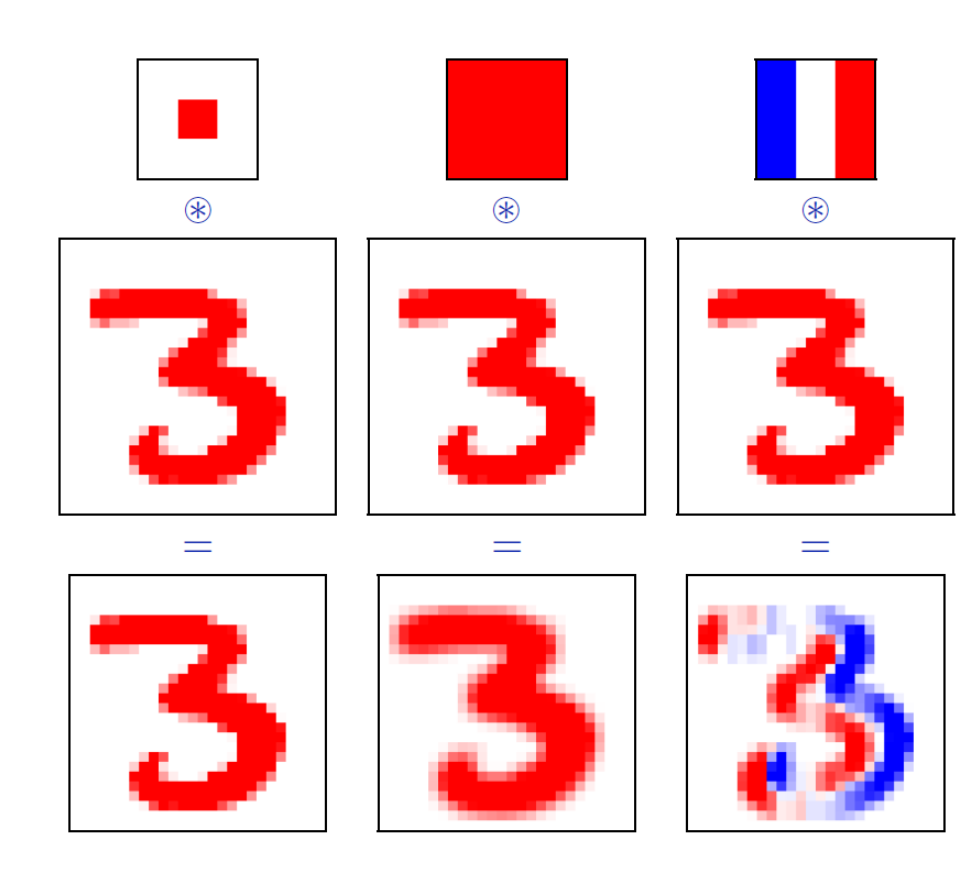
6) Pytorch – Convolutionals



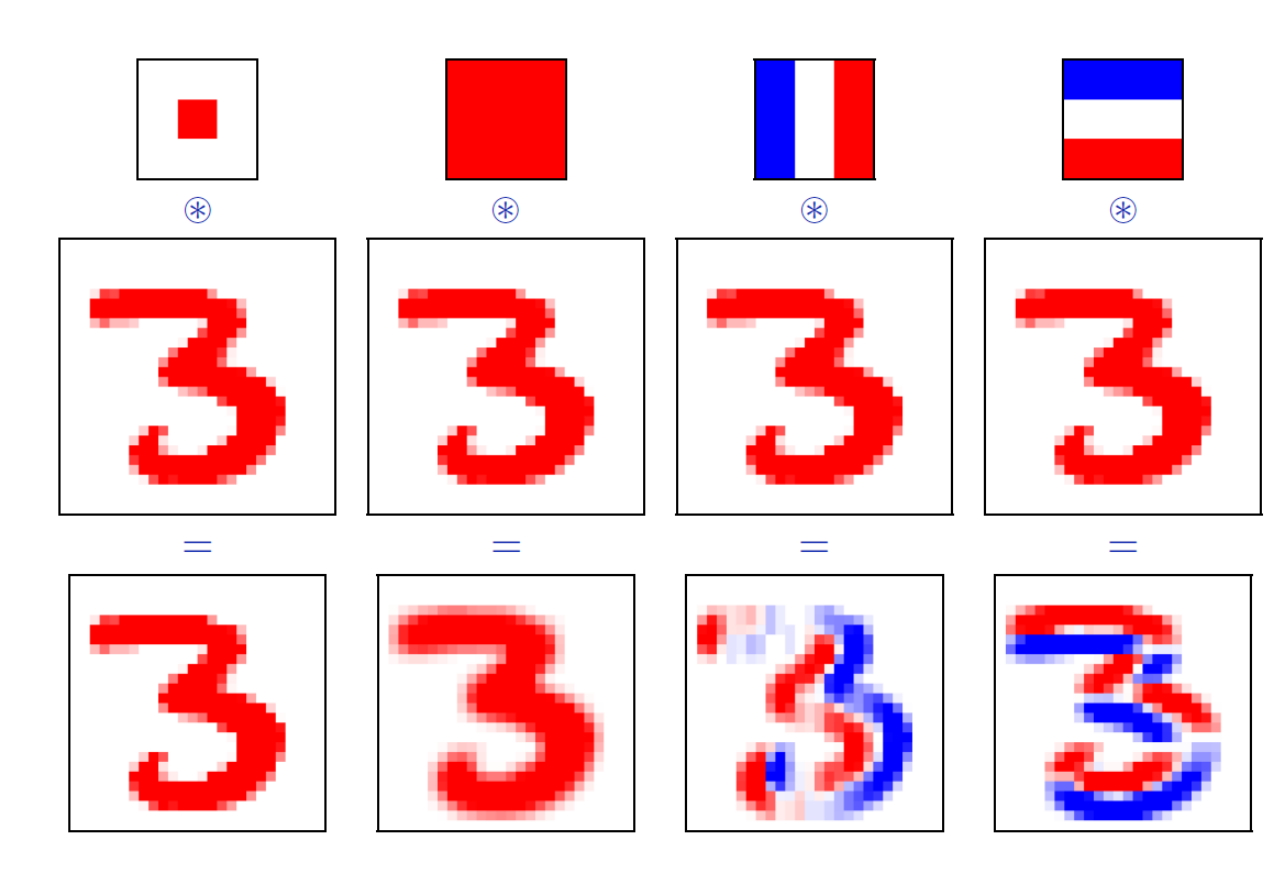
6) Pytorch – Convolutionals



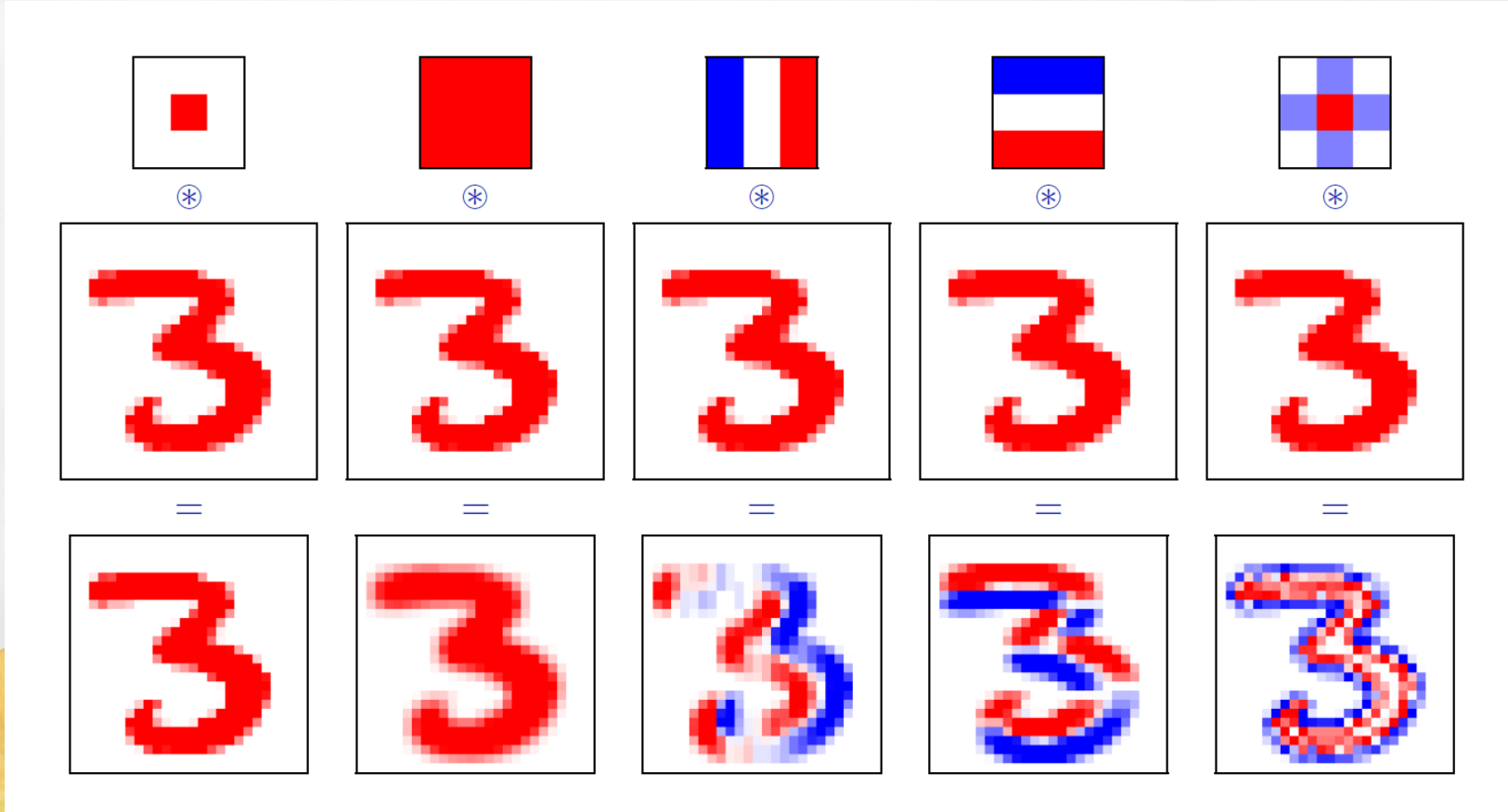
6) Pytorch – Convolutionals



6) Pytorch – Convolutionals



6) Pytorch – Convolutionals



6) Pytorch – Convolutionals

class torch.nn.Conv2d(in_channels, out_channels,
kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

- convolution을 Module로 생성 시 kernel과 bias를 매개변수로 적절히 초기화 하면서 래핑 가능함.

- 커널 크기는 (h, w) 로 표현하거나, 상수 k 값으로 표현 할 수 있는데, 이는 (k, k) 임.

```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.empty(117, 4, 10, 3).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

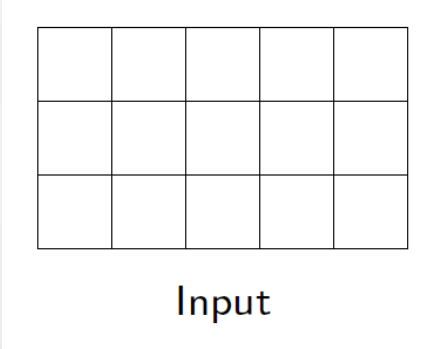
7) Pytorch – Padding and stride

- Convolutions은 두 가지 추가적인 표준 매개변수를 지님:

- **padding** 은 입력 주변에 0으로 채우는 frame을 의미하고,
- **stride** 신호를 가로 질러 kernel이 움직이는 step 크기를 지칭함.

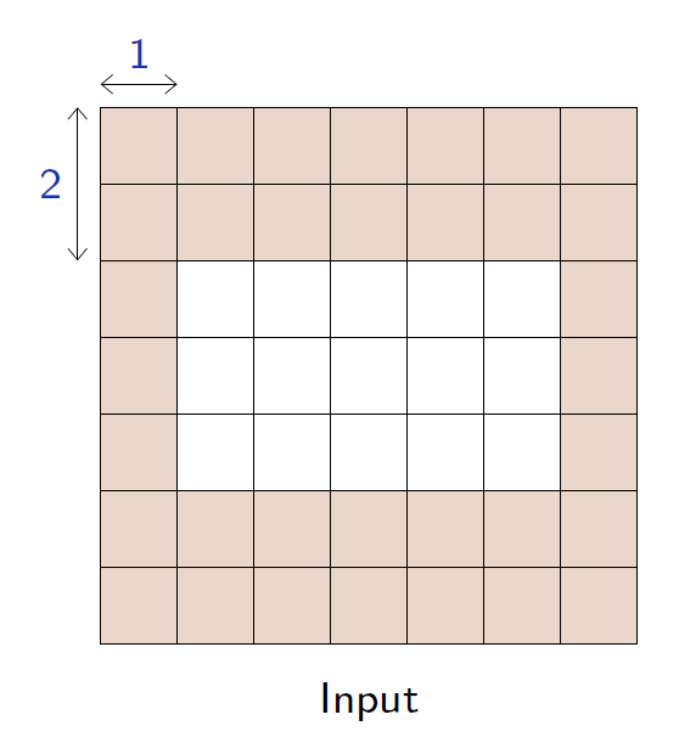
7) Pytorch – Padding and stride

- Here with $C \times 3 \times 5$ as input



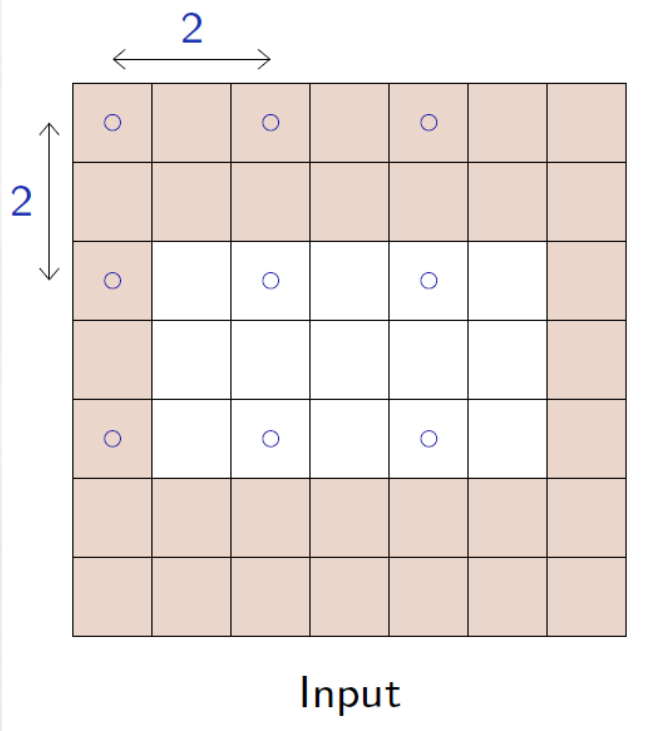
7) Pytorch – Padding and stride

- Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$



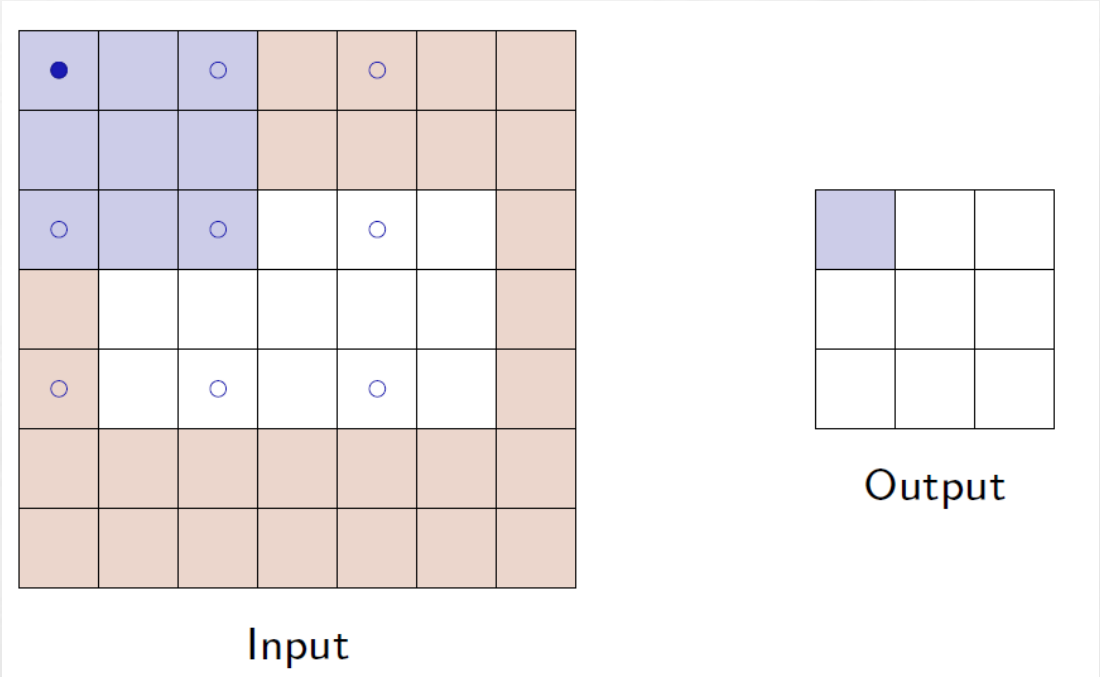
7) Pytorch – Padding and stride

- Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$



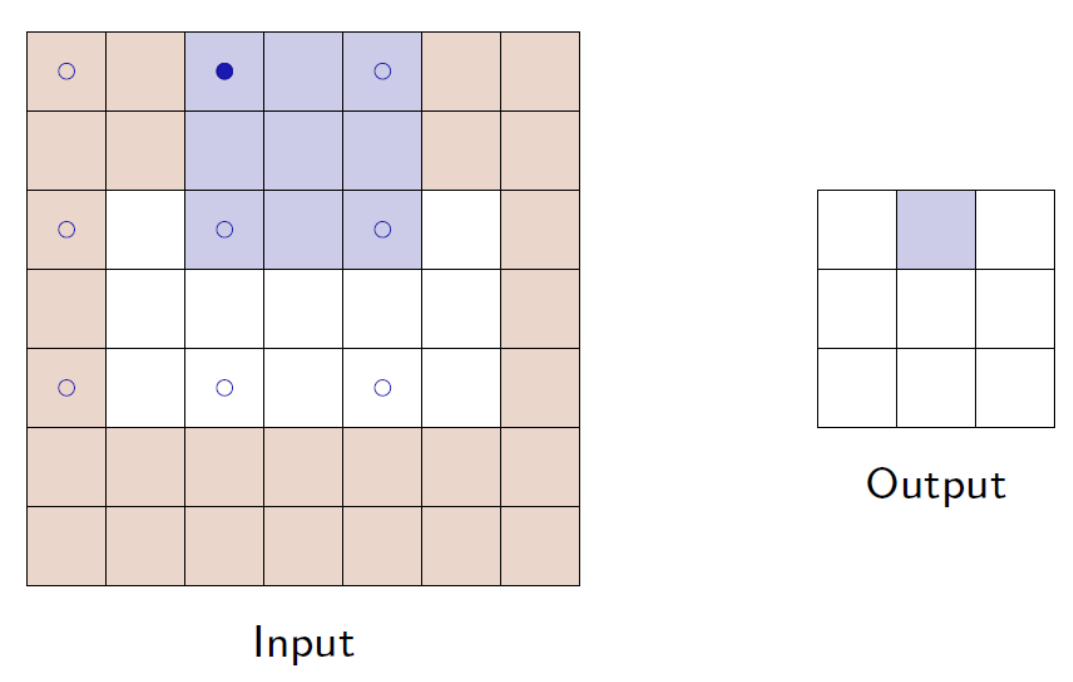
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



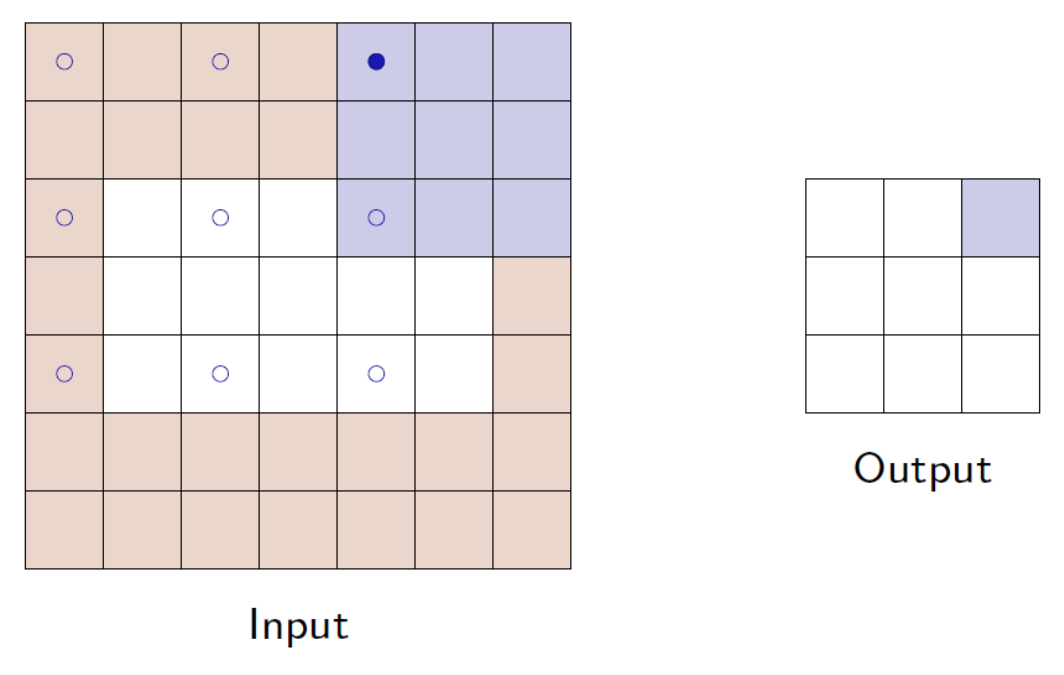
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



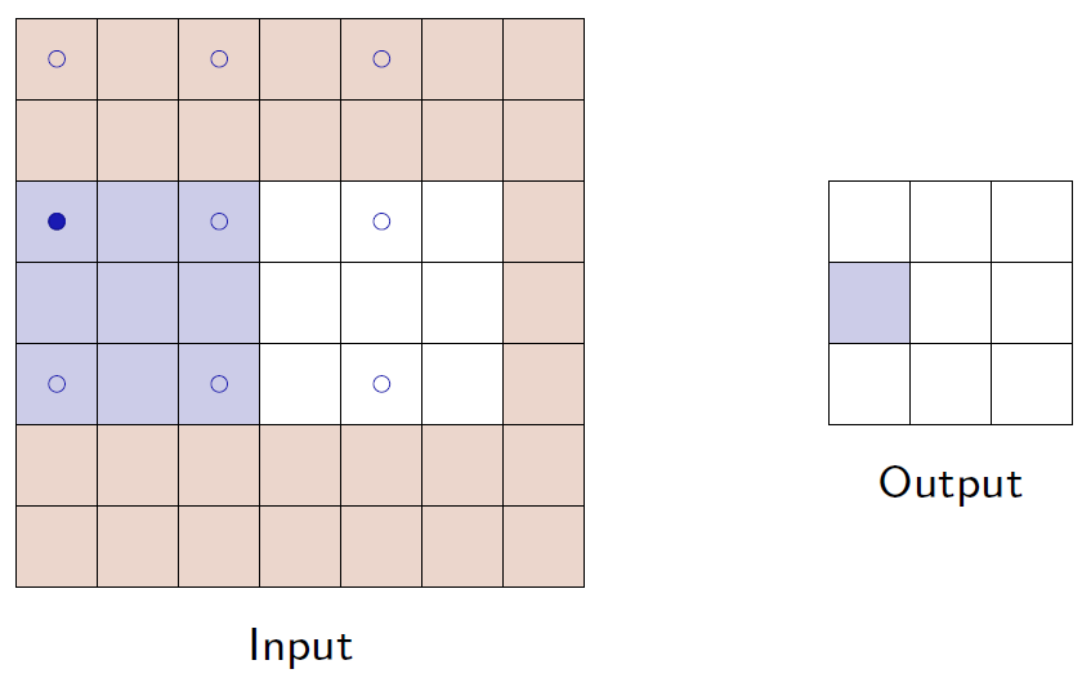
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



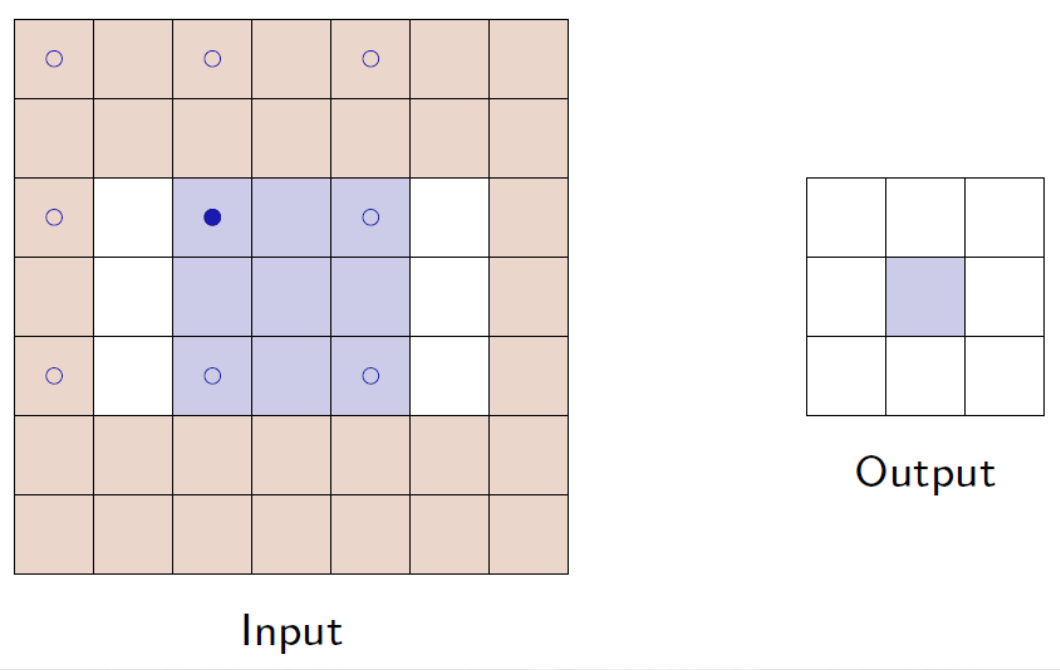
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



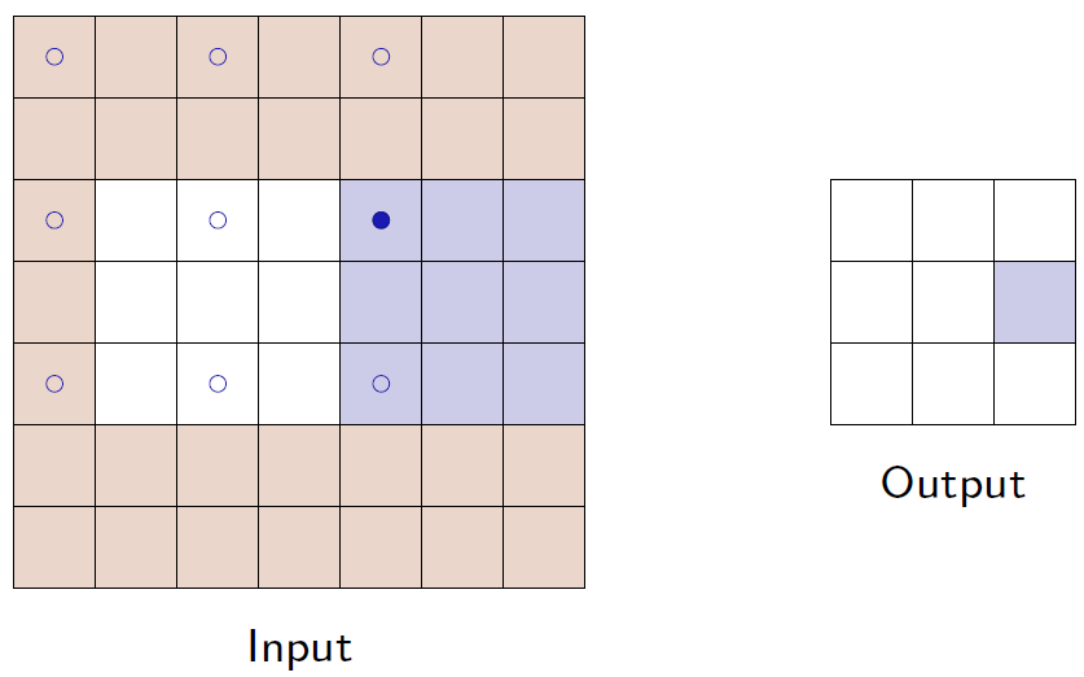
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



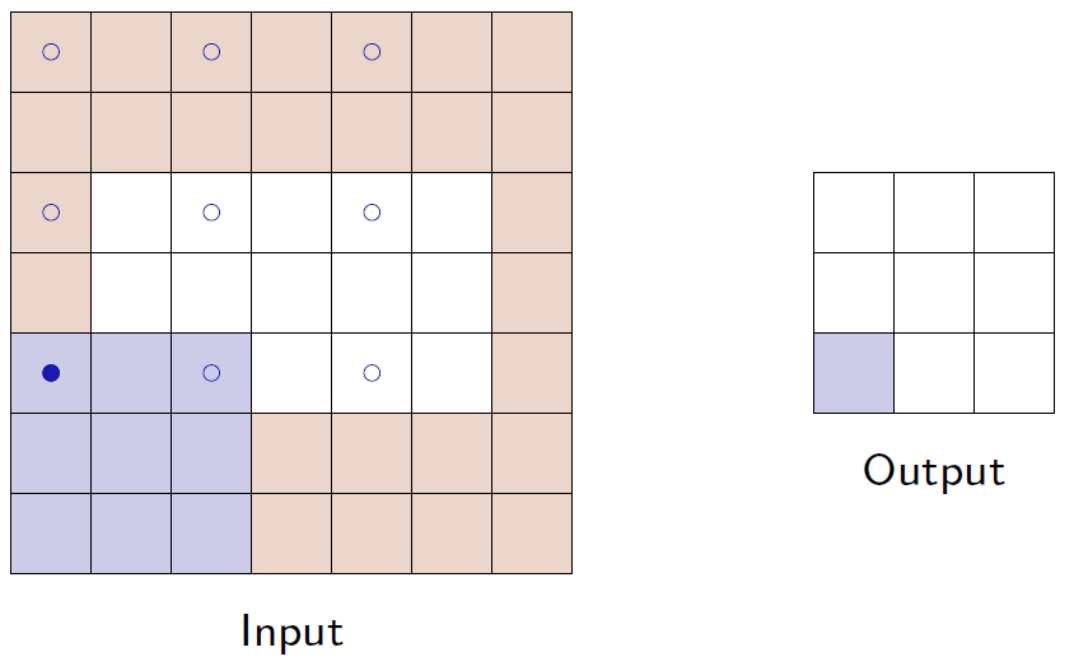
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



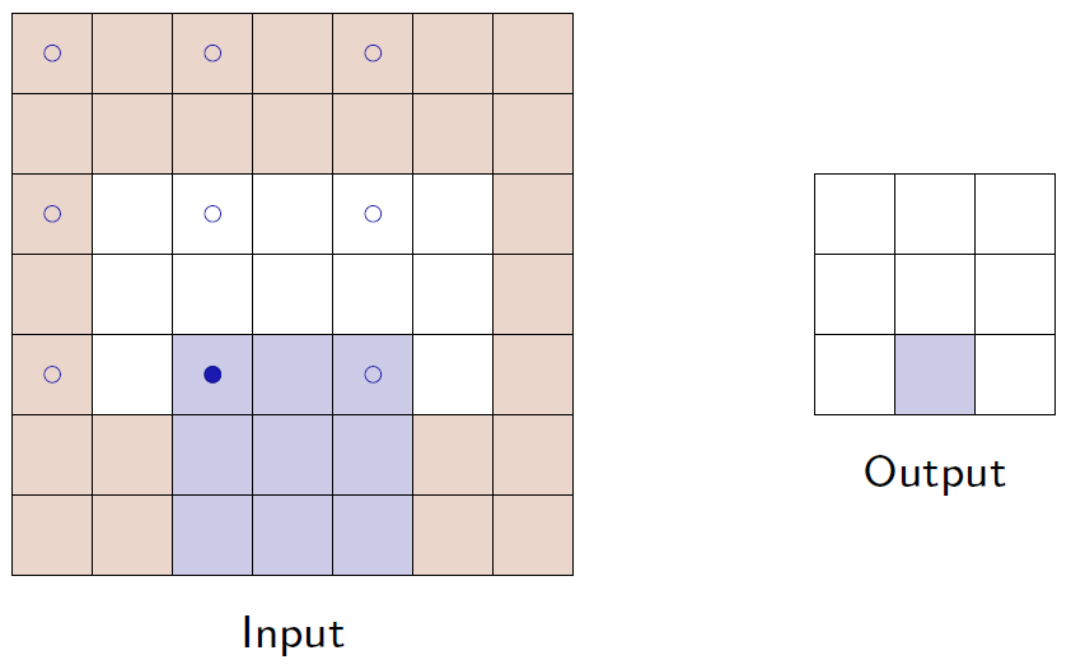
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



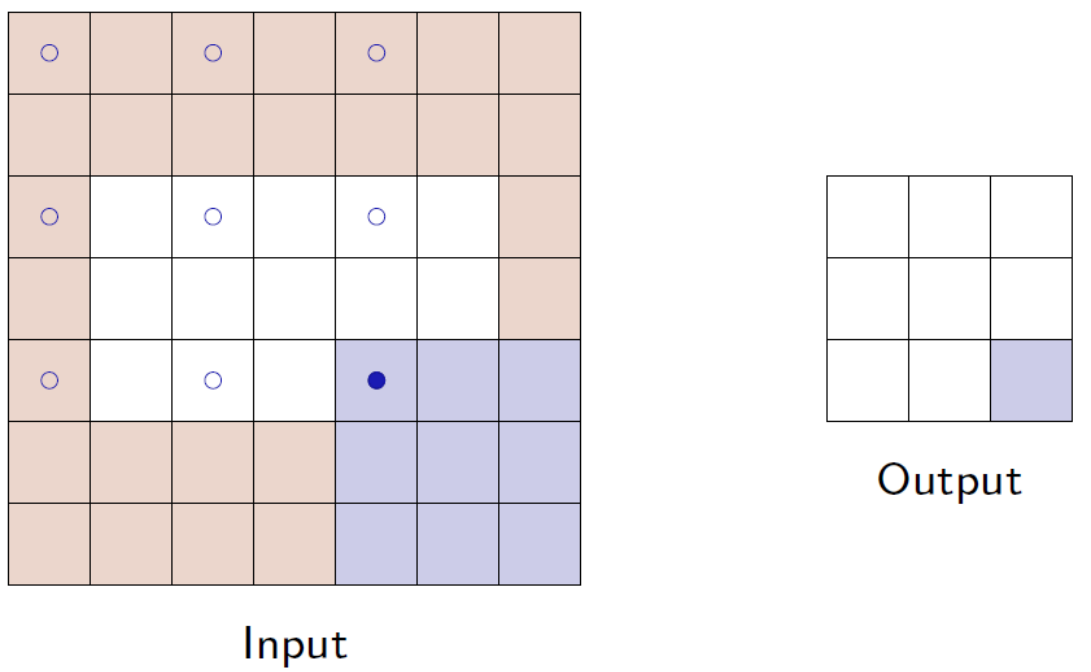
7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$

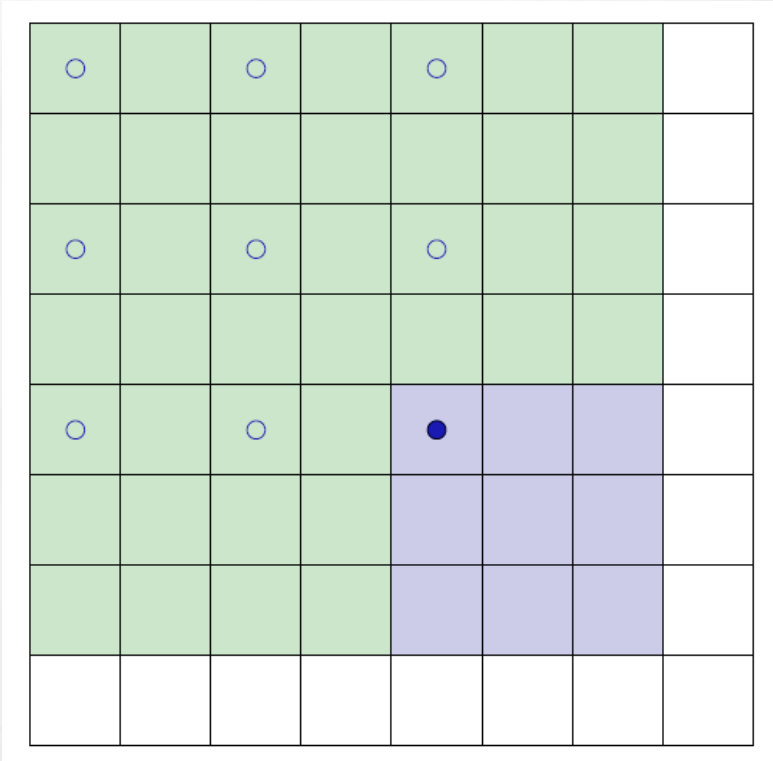


7) Pytorch – Padding and stride

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$ and a kernel of size $C \times 3 \times 3$



7) Pytorch – Padding and stride



! stride가 1보다 큰 convolution은 입력 맵을 완전히 덮지 않을 수 있으므로, 일부 입력 값을 무시할 수 있음.

이

8) Dilated Convolution

- Convolution 연산은 필터 지원을 확장을 조절하는

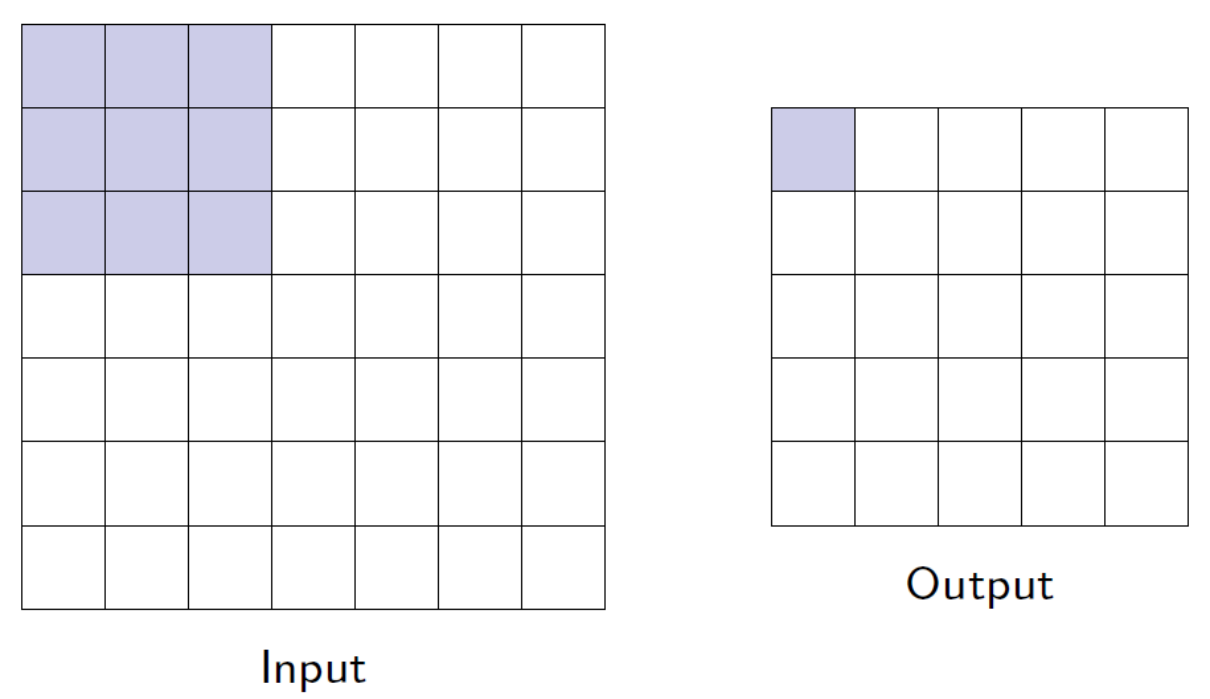
dilation이라는 표준 매개 변수를 허용함. (Yu and Koltun, 2015).

- 표준 convolution의 경우 1이지만 더 클 수 있음.

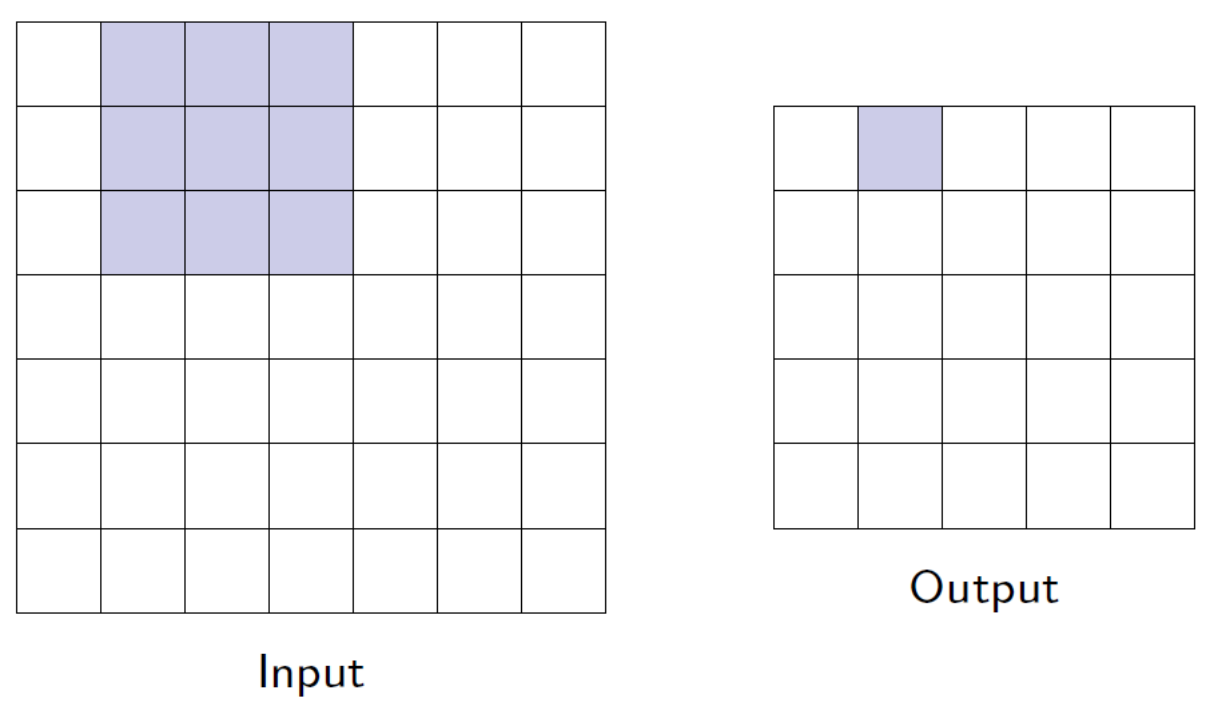
이 경우 결과 연산은 규칙적으로 희소 화 된 필터가 있는 convolution으로 구상 될 수 있음.

- 이 개념은 신호 처리에서 비롯됨. 여기서 알고리즘은 trous라고 하며, 따라서 용어는 때때로 " convolution a trous"로 사용됨.

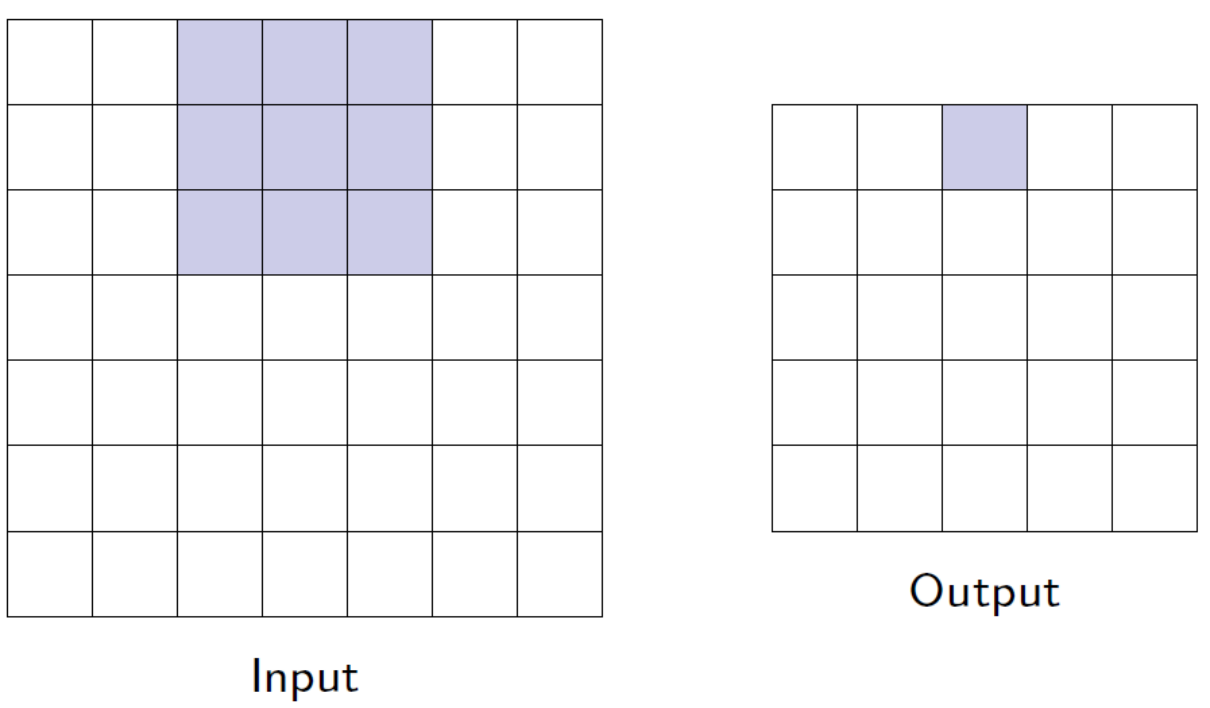
8) Dilated Convolution



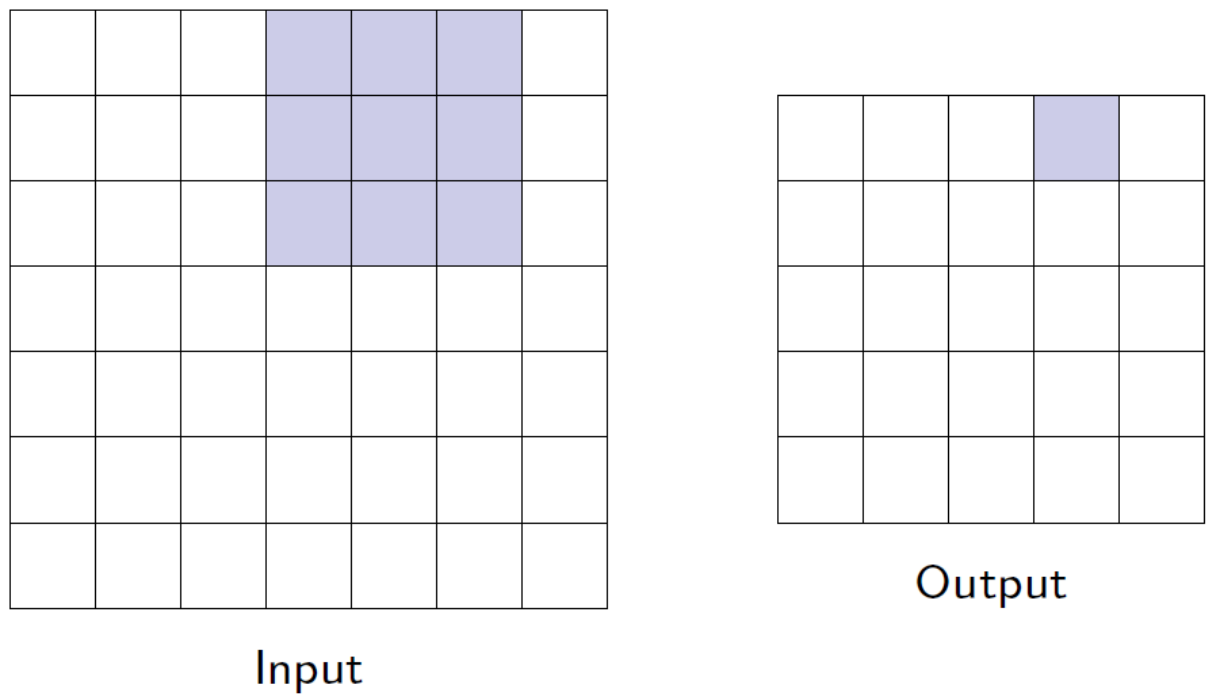
8) Dilated Convolution



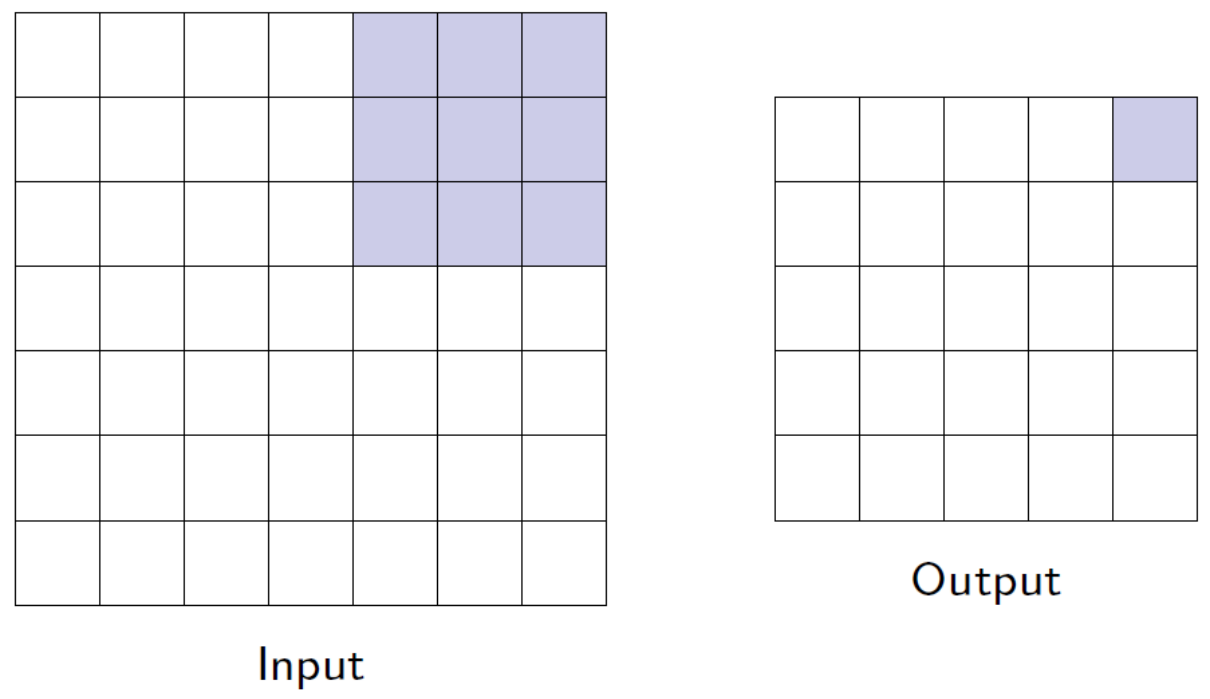
8) Dilated Convolution



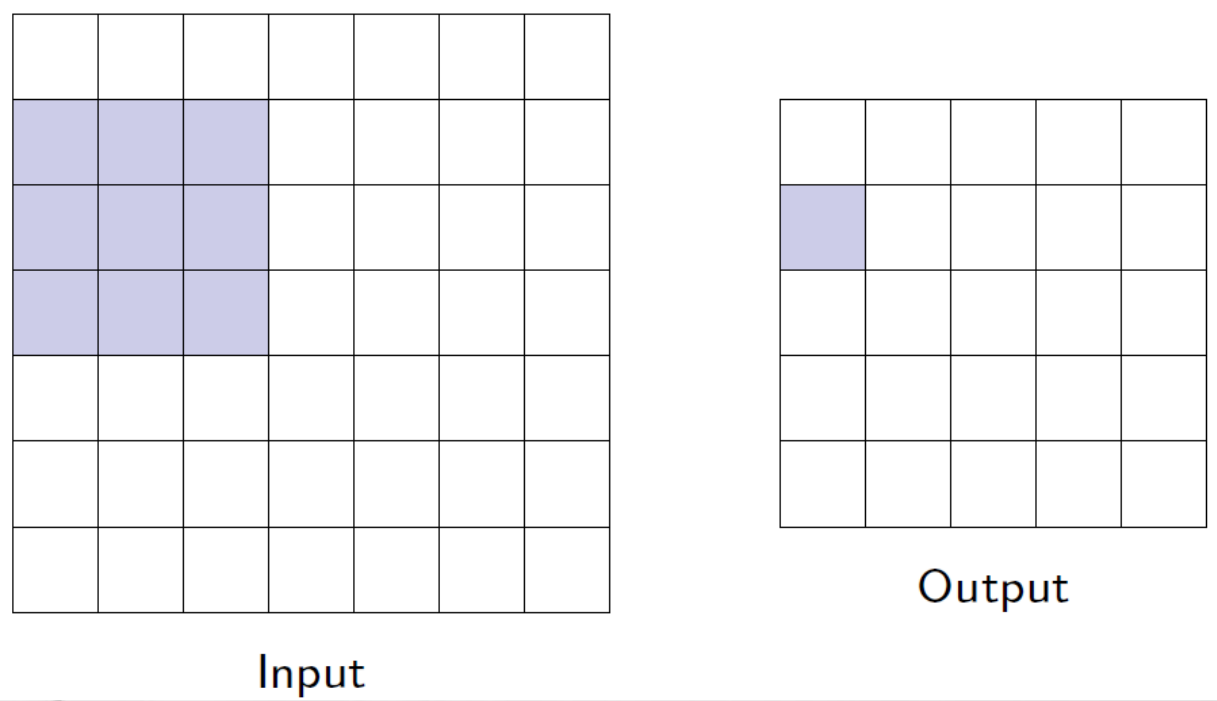
8) Dilated Convolution



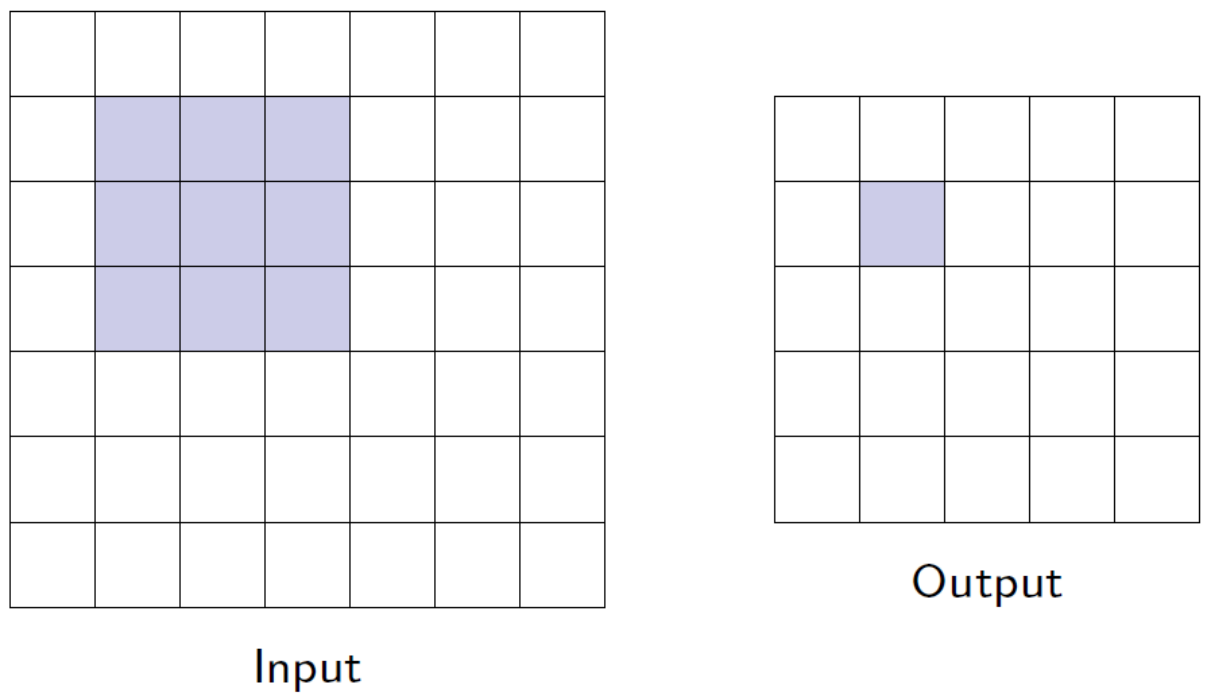
8) Dilated Convolution



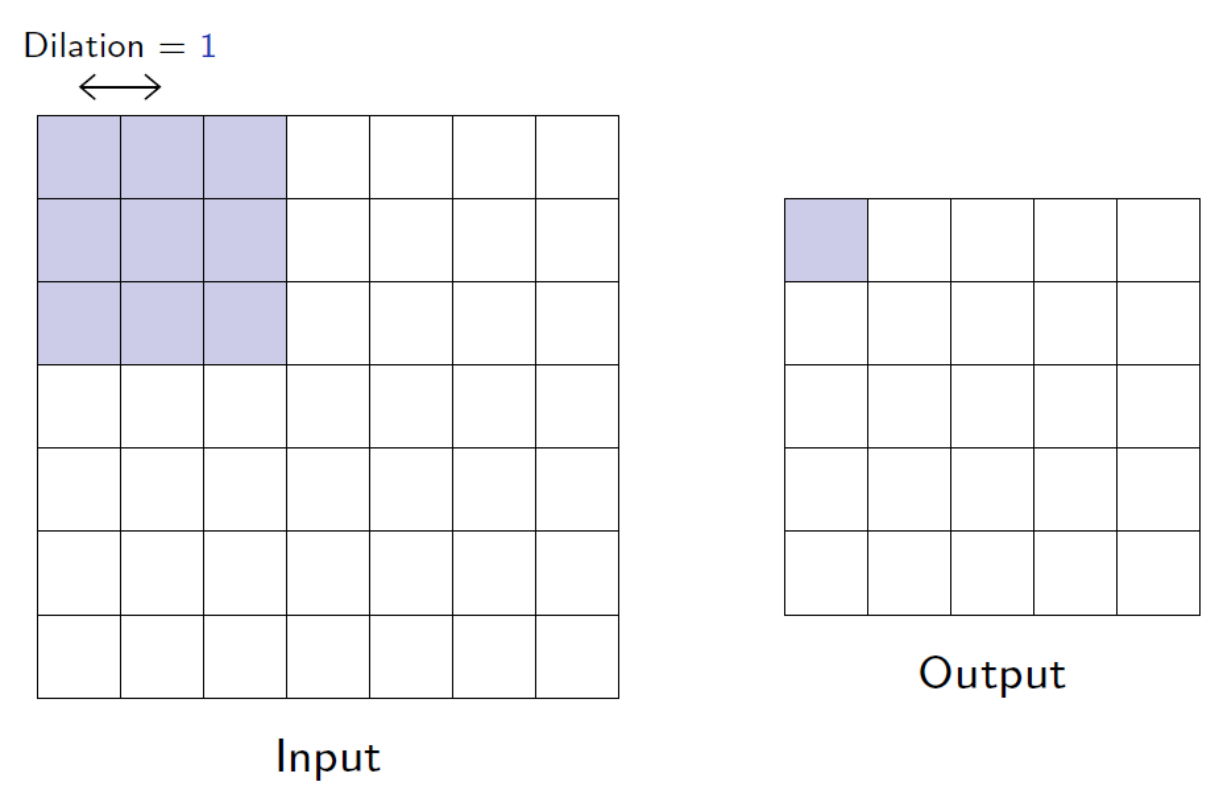
8) Dilated Convolution



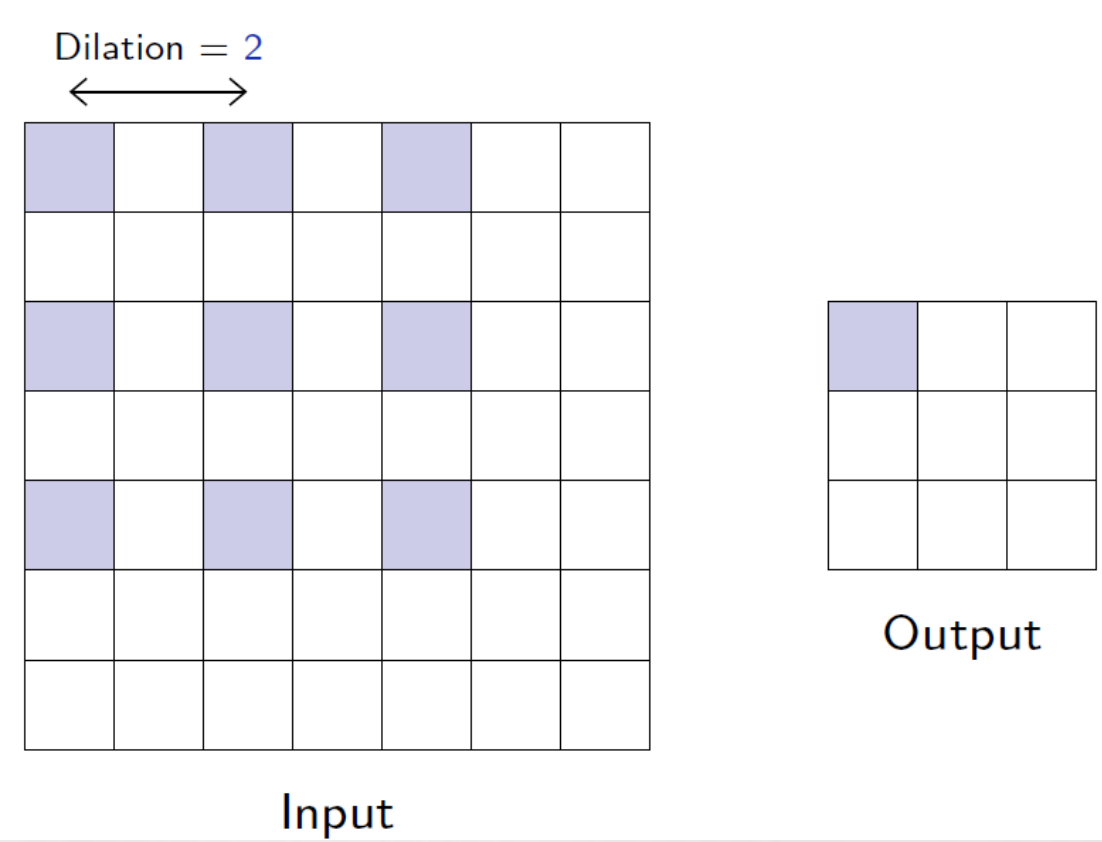
8) Dilated Convolution



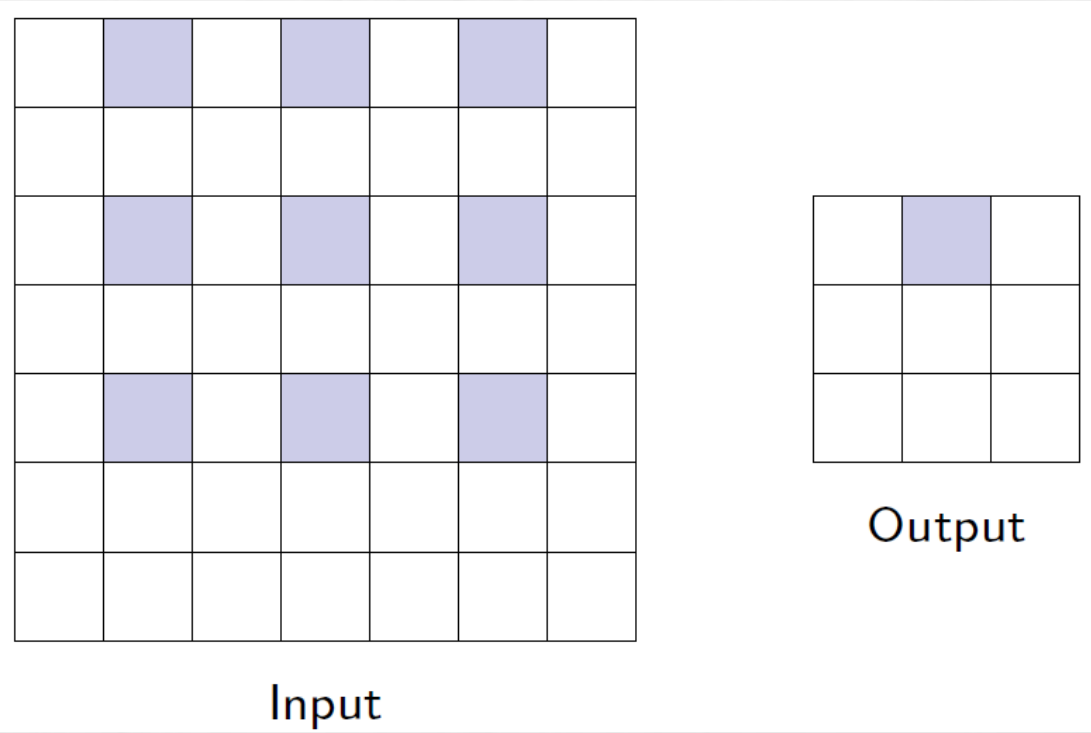
8) Dilated Convolution



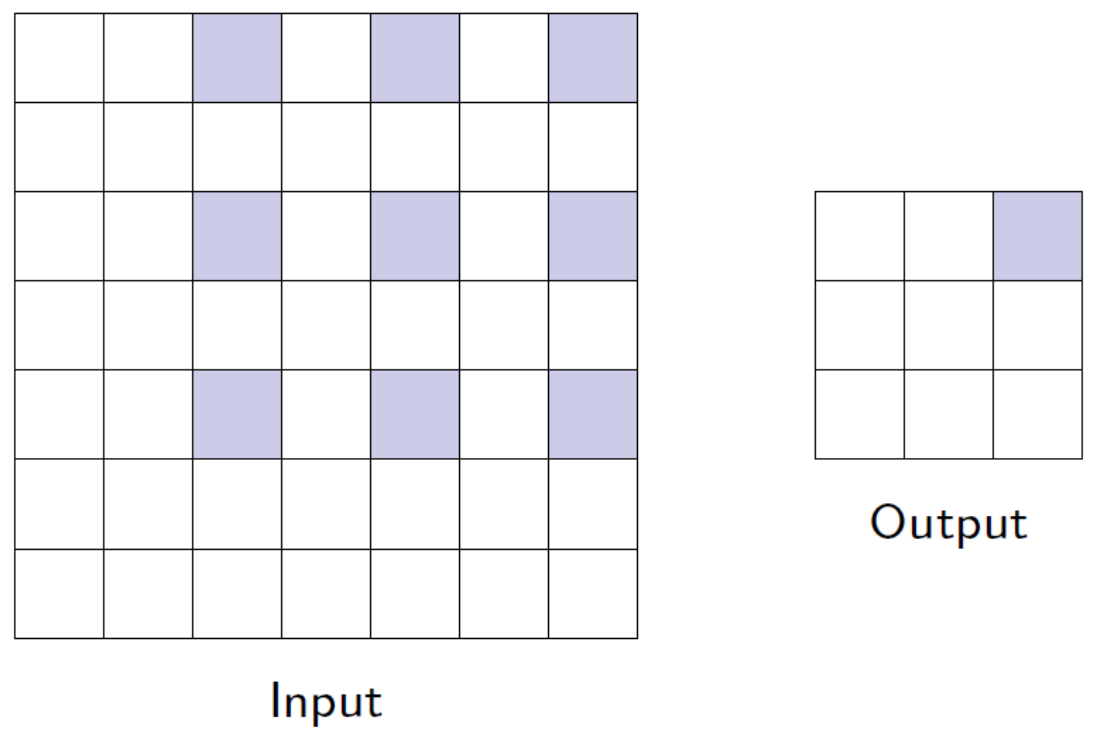
8) Dilated Convolution



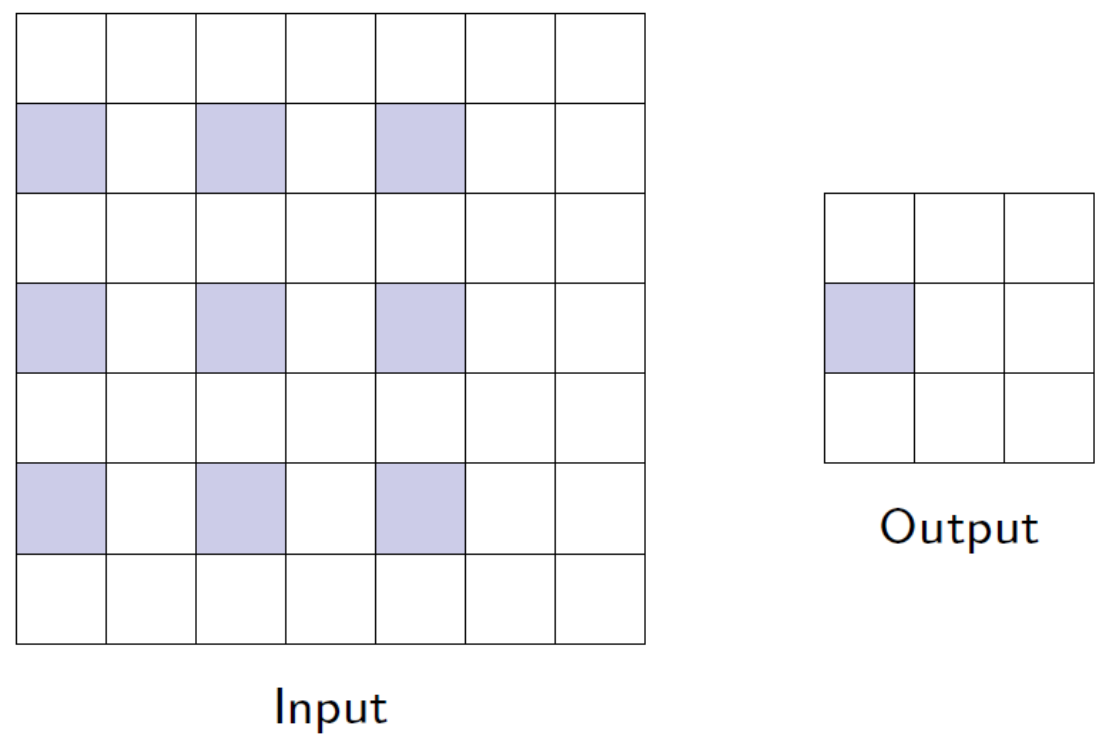
8) Dilated Convolution



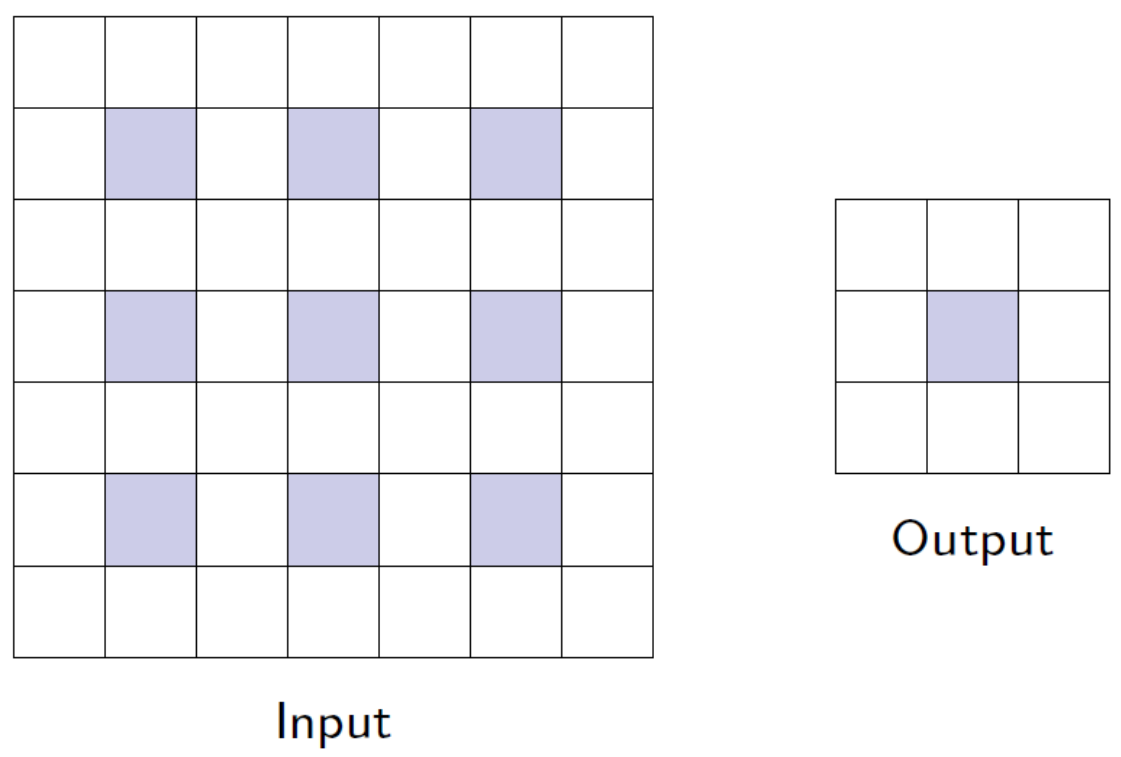
8) Dilated Convolution



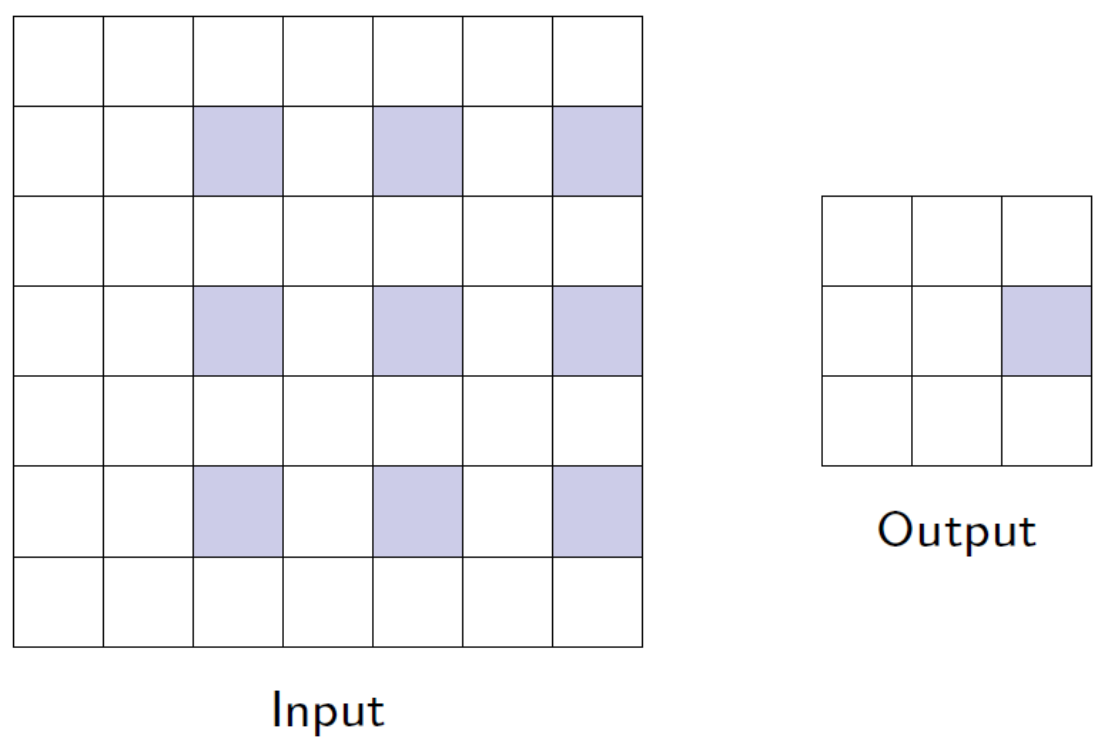
8) Dilated Convolution



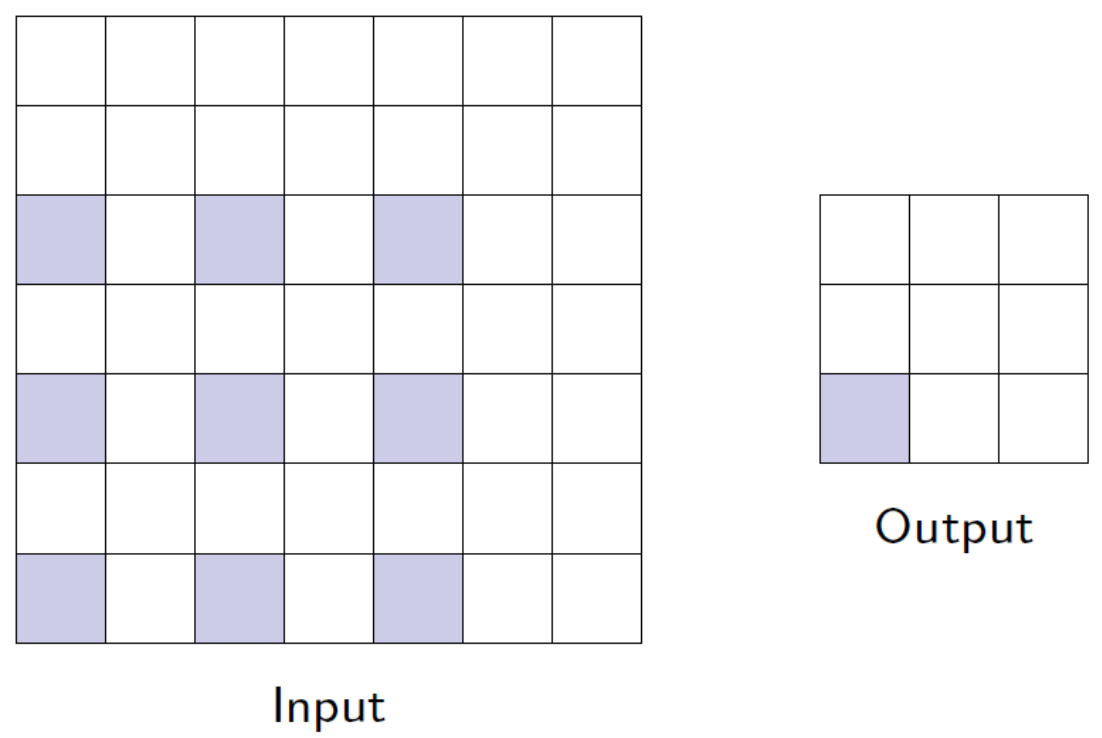
8) Dilated Convolution



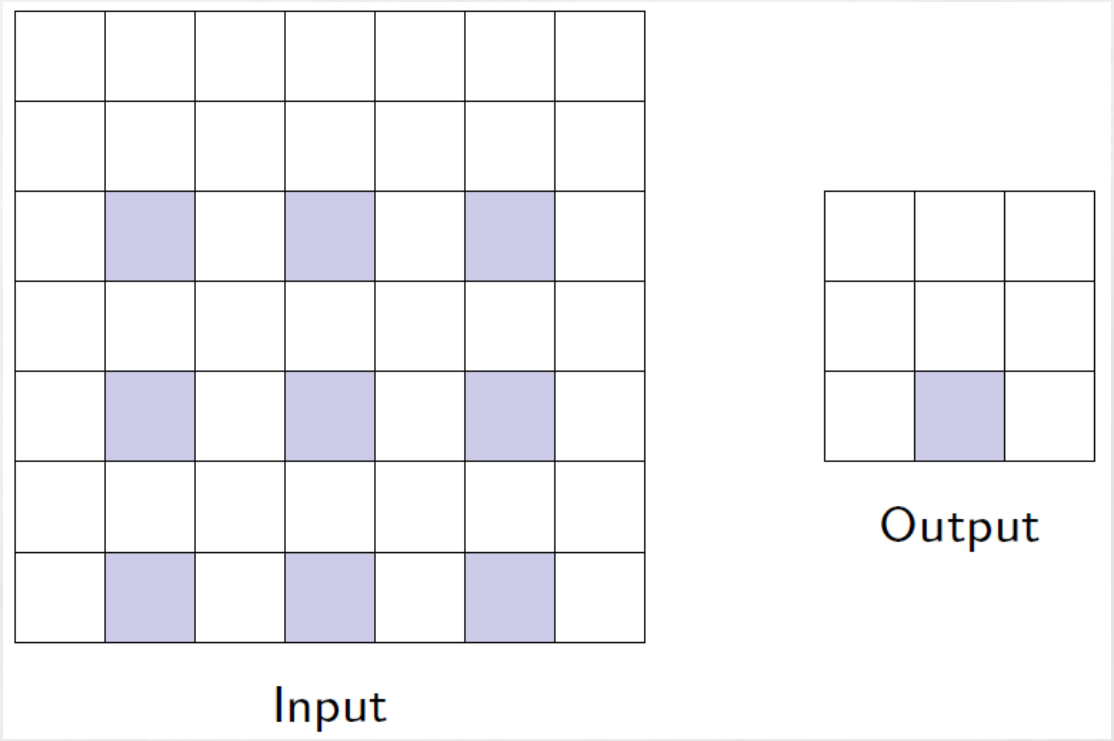
8) Dilated Convolution



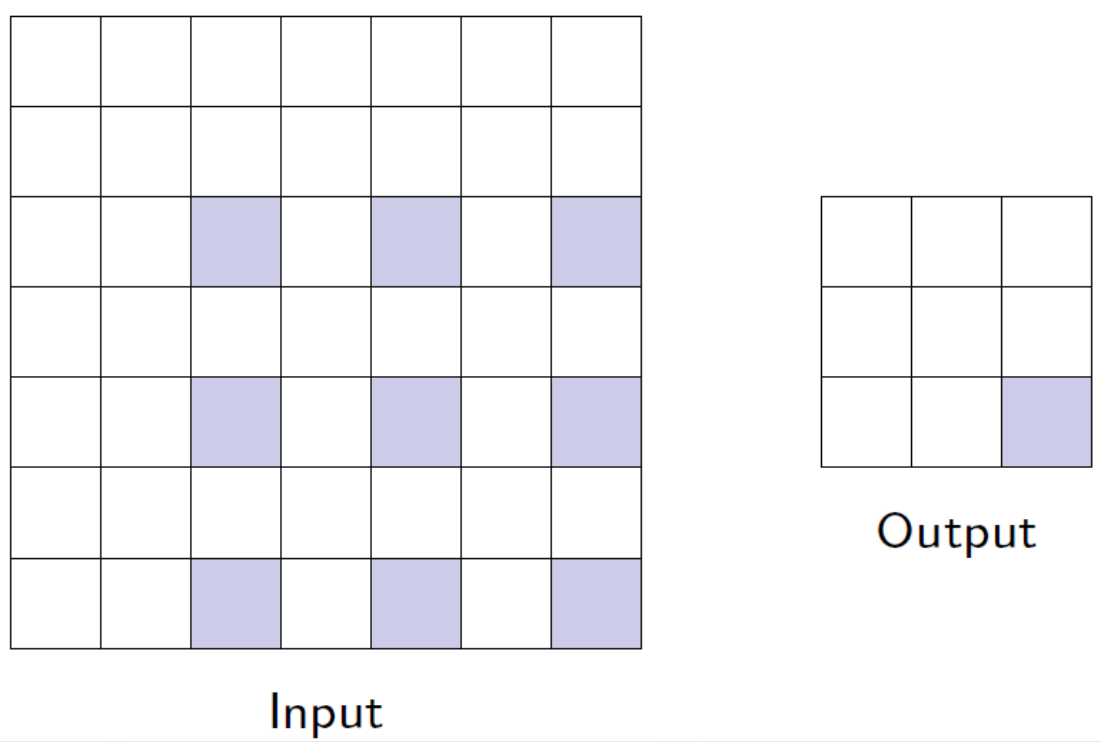
8) Dilated Convolution



8) Dilated Convolution



8) Dilated Convolution



8) Dilated Convolution

- 크기가 k 이고 dilation d A인 1d kernel이 있는 convolution은 0이 아닌 계수가 k 개인 필터 크기가 $1 + (k - 1)d$ 인 convolution으로 해석 될 수 있음.

- $k = 3$ 이고 $d = 4$인 경우, 입력 map과 출력 map의 크기 차이는 $1 + (3 - 1)4 - 1 = 8$.

```
In [63]:
1 x = torch.empty(1, 1, 20, 30).normal_()
2 l = nn.Conv2d(1, 1, kernel_size=3, dilation=4)
3 l(x).size()

Out[63]: torch.Size([1, 1, 12, 22])
```

8) Dilated Convolution

- dilation이 1보다 크면 매개 변수의 수를 늘리지 않고 단위의 receptive field 크기가 증가함.

- stride 또는 dilation이 1보다 큰 Convolutions는 마지막 classification decision에서 activation map 크기를 줄임.

- 그러한 네트워크의 단순성이 지니는 장점은:
  - 비선형 연산이 활성 함수에서만 이뤄진다는 점과,
  - 하나의 결과를 만드는 여러 활성함수를 결합하는 연산이 단지 선형 계층에서만 이뤄진다는 점임.

9) Creating a Module

- 이제 처음부터 첫 번째 convolutional network를 구축하는 데 필요한 브릭이 있음. 마지막 기술 포인트는 계층들(layer) 사이의 Tensor 모양임.
- convolutional layers 와 pooling layers는 모두 샘플의 입력을 배치로 사용하며 각 layer는 그 자체가 크기 $C \times H \times W$ 의 3D 텐서임.
- 출력은 같은 구조를 지니며, tensor들은 마지막 완전 연결 계층(fully connected layer)로 전달되기 전에 tensor형태가 변경되어 야 함.

In [67]:
```
1 from torchvision.datasets import MNIST
2
3 mnist = MNIST('./data/mnist/', train=True, download=True)
4 d = mnist.data
5 d.size()
```

Out[67]: torch.Size([60000, 28, 28])

In [65]:
```
1 x = d.view(d.size(0), 1, d.size(1), d.size(2))
2 x.size()
```

Out[65]: torch.Size([60000, 1, 28, 28])

In [66]:
```
1 x = x.view(x.size(0), -1)
2 x.size()
```

Out[66]: torch.Size([60000, 784])

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$ `nn.Conv2d(1, 32, kernel_size=5)` $32 \times 24 \times 24$	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		
`F.relu(.)`	0	0
200		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		
`F.relu(.)`	0	0
200		
`nn.Linear(200, 10)`	$10 \times (200 + 1) = 2{,}010$	$10 \times 200 = 2{,}000$
10		

9) Creating a Module

- A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		
`F.relu(.)`	0	0
200		
`nn.Linear(200, 10)`	$10 \times (200 + 1) = 2{,}010$	$10 \times 200 = 2{,}000$
10		

- Total 105,506 parameters and 1,333,200 products for the forward pass.

9) Creating a Module

- PyTorch 단순 architecture를 만들 수 있도록 sequential container module인 torch.nn.Sequential 지원함.

- 예를 들어, 입력이 10차원이고 출력이 2차원인 MLP 에 대해, ReLU 활성 함수와 100차원과 50차원의 두 은닉 계층은 다음과 같이 표현 가능함:

```
model = nn.Sequential(
nn.Linear(10, 100), nn.ReLU(),
nn.Linear(100, 50), nn.ReLU(),
nn.Linear(50, 2)
);
```

! 그러나, 산업계에 실제 응용되는 모델의 연산 복잡도를 고려할 때, 가장 좋은 접근 방법은 torch.nn.Module 의 sub-class를 구성하는 것임

9) Creating a Module

- Module 생성을 위해서는 base class를 상속 후, constructor __init__(self, ...) 과 the forward pass forward(self, x) 을 구현 해야 함.

In [68]:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

9) Creating a Module

- torch.nn.Module 로 부터 상속을 함으로써 superclass에 구현된 많은 mechanism을 상속받게 됨.
- 먼저, (...) 연산자는 forward(...) 메서드 호출을 통해 재 정의되고, 추가 연산들을 실행함. The forward pass 는 이 연산자를 통해서 실행 되어야 하므로 직접 forward 호출하지 않도록 한다.
- 앞서 정의한 Net 클래스에 대해 우리는 단지 다음과 같이 정의 가능함

In [69]:
```
1 model = Net()
2 input = torch.empty(12, 1, 28, 28).normal_()
3 output = model(input)
4 print(output.size())
```

```
torch.Size([12, 10])
```

9) Creating a Module

- 또한 클래스 속성으로 더해진 모든 매개변수(Parameters)들은 Module.parameters()를 통해 확인 가능 함.

In [68]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

In [69]:
```
model = Net()
input = torch.empty(12, 1, 28, 28).normal_()
output = model(input)
print(output.size())
```

```
torch.Size([12, 10])
```

In [70]:
```
for k in model.parameters():
    print(k.size())
```

In [70]:
```
for k in model.parameters():
    print(k.size())
```

```
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```

9) Creating a Module

! dictionaries 혹은 행렬에 더해진 매개변수는 볼 수 없음

```
class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = [ nn.Linear(543, 21) ]

model = Buggy()

for k in model.parameters():
    print(k.size())
```

prints

```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
```

9) Creating a Module

! torch.nn.ModuleList는 모듈들을 더하는 단순 옵션으로 PyTorch 기계(?)에 의해 다루는 module들 list임.

```
class AnotherNotBuggy(nn.Module):
    def __init__(self):
        super(AnotherNotBuggy, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = nn.ModuleList()
        self.other_stuff.append(nn.Linear(543, 21))

model = AnotherNotBuggy()

for k in model.parameters():
    print(k.size())
```

prints

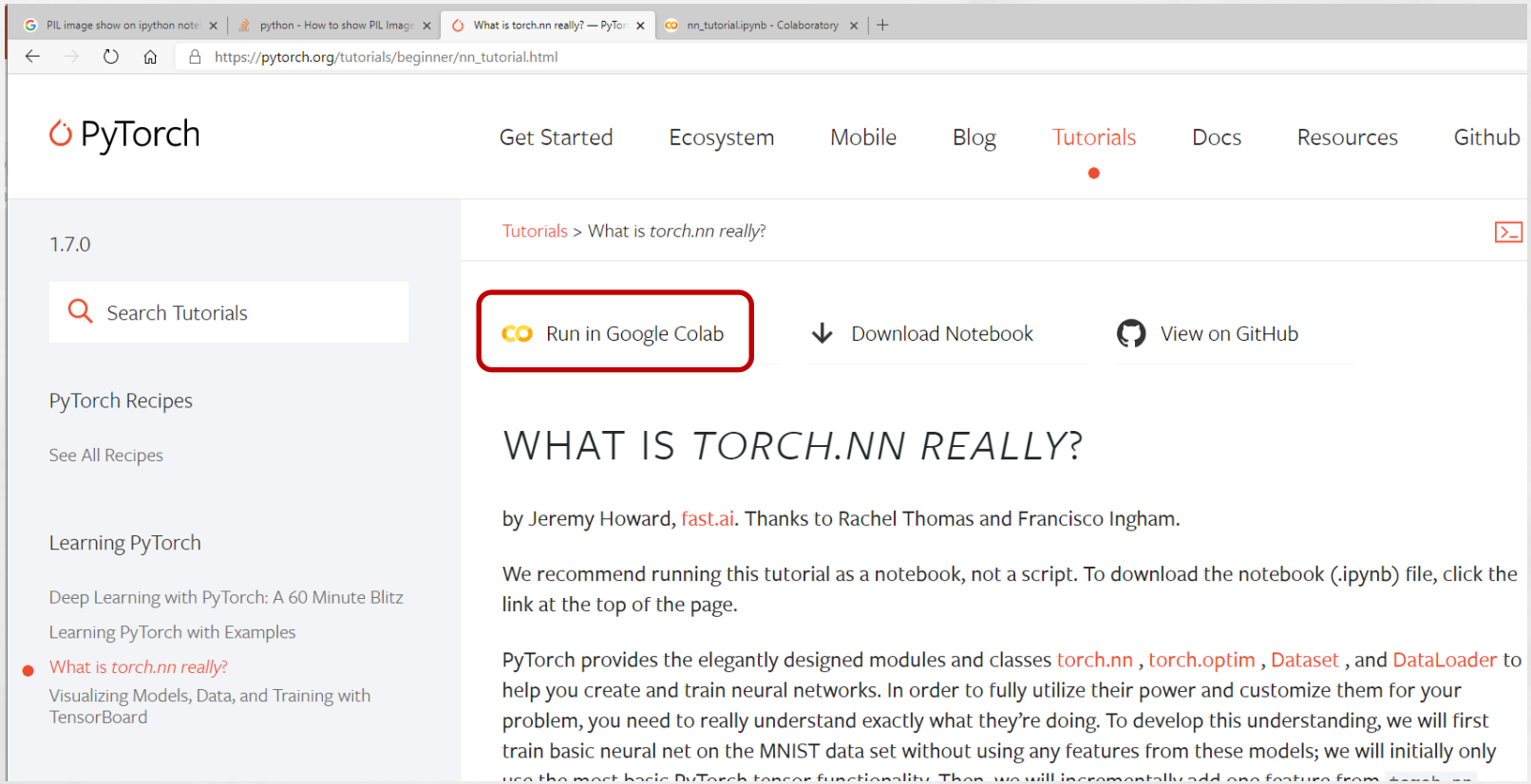
```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
torch.Size([21])
```

9) Creating a Module

- autograd 에 호환되는 연산자들을 이용하면, back ward pass는 자동으로 구현됨.
- 이는 경사 하강 법(gradient descent)으로 매개변수들을 최적화하는 데 있어 매우 중요함.

7) Pytorch 실습 on Google CoLab

https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/5ddab57bb7482fbcc76722617dd47324/nn_tutorial.ipynb



감사합니다.