Convolutional Neural Networks (CNN)

Industrial AI Lab.

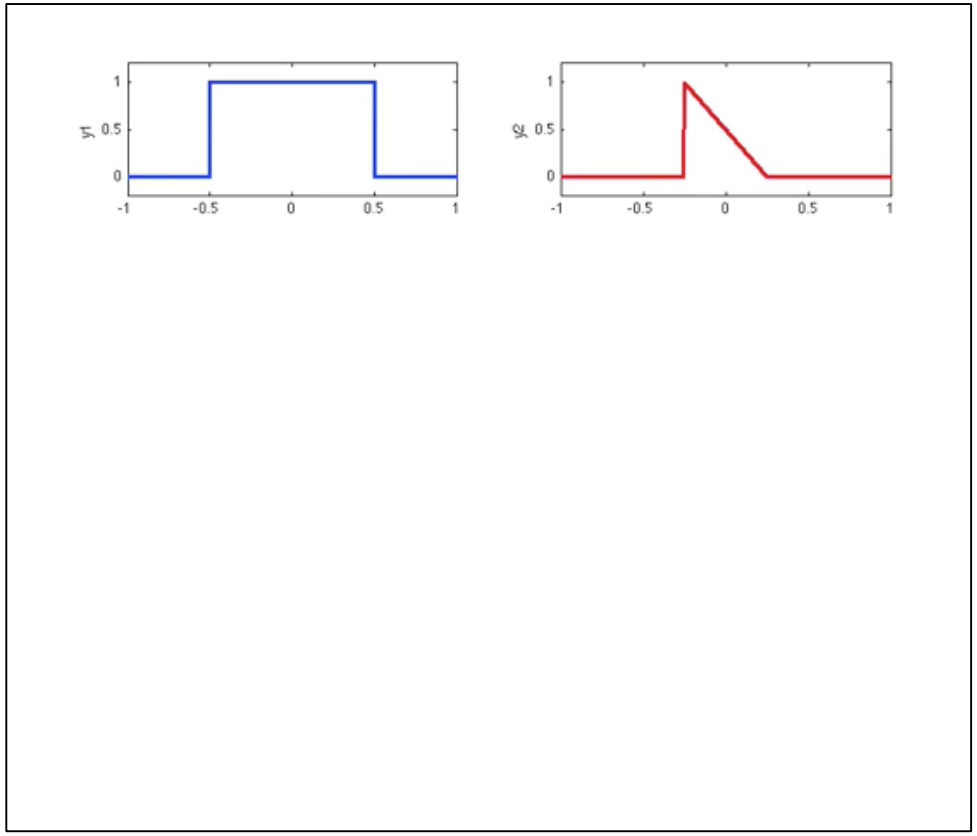
Prof. Seungchul Lee

Convolution

Convolution

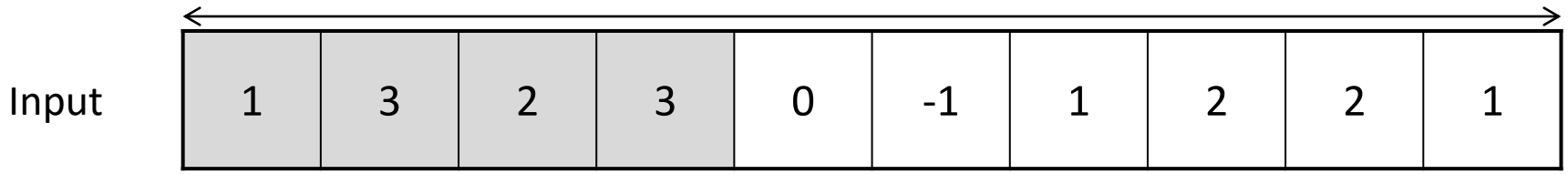
- Integral (or sum) of the product of the two signals after one is reversed and shifted
- Cross correlation and convolution

$$y[n] = \sum_{m=-\infty}^{\infty} h[n-m]\,x[m] = x[n] * h[n]$$



1D Convolution

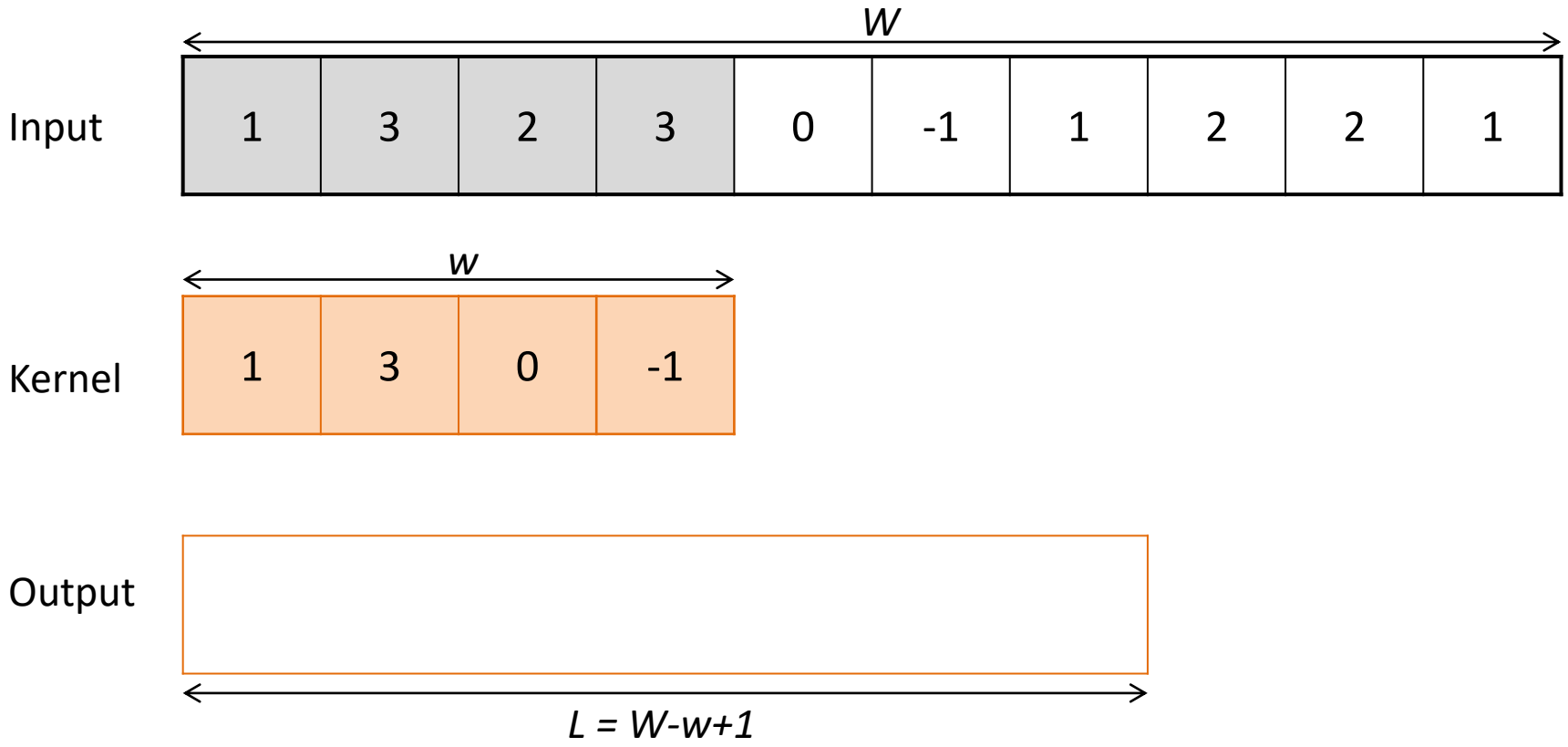
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

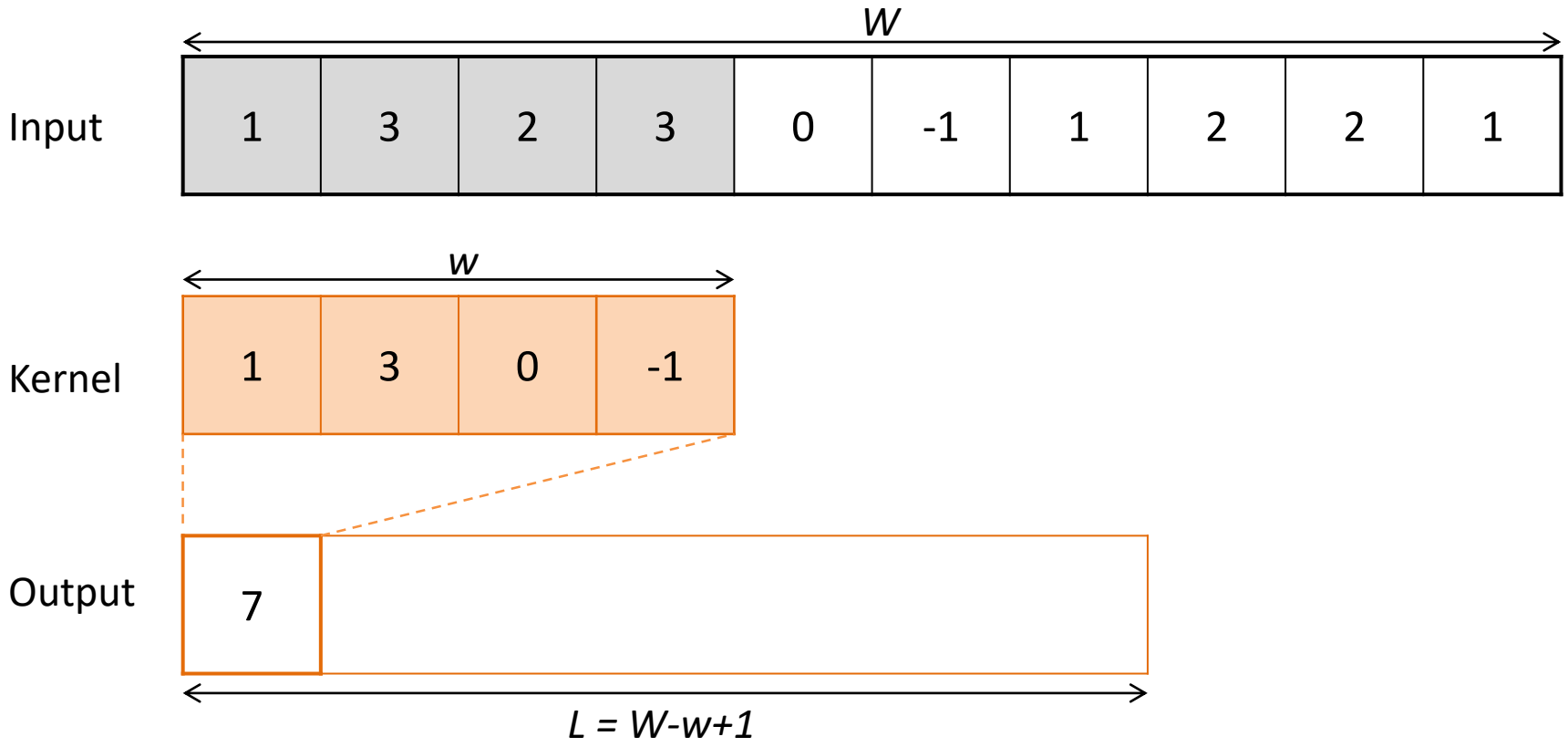
1D Convolution

- (actually cross-correlation)



1D Convolution

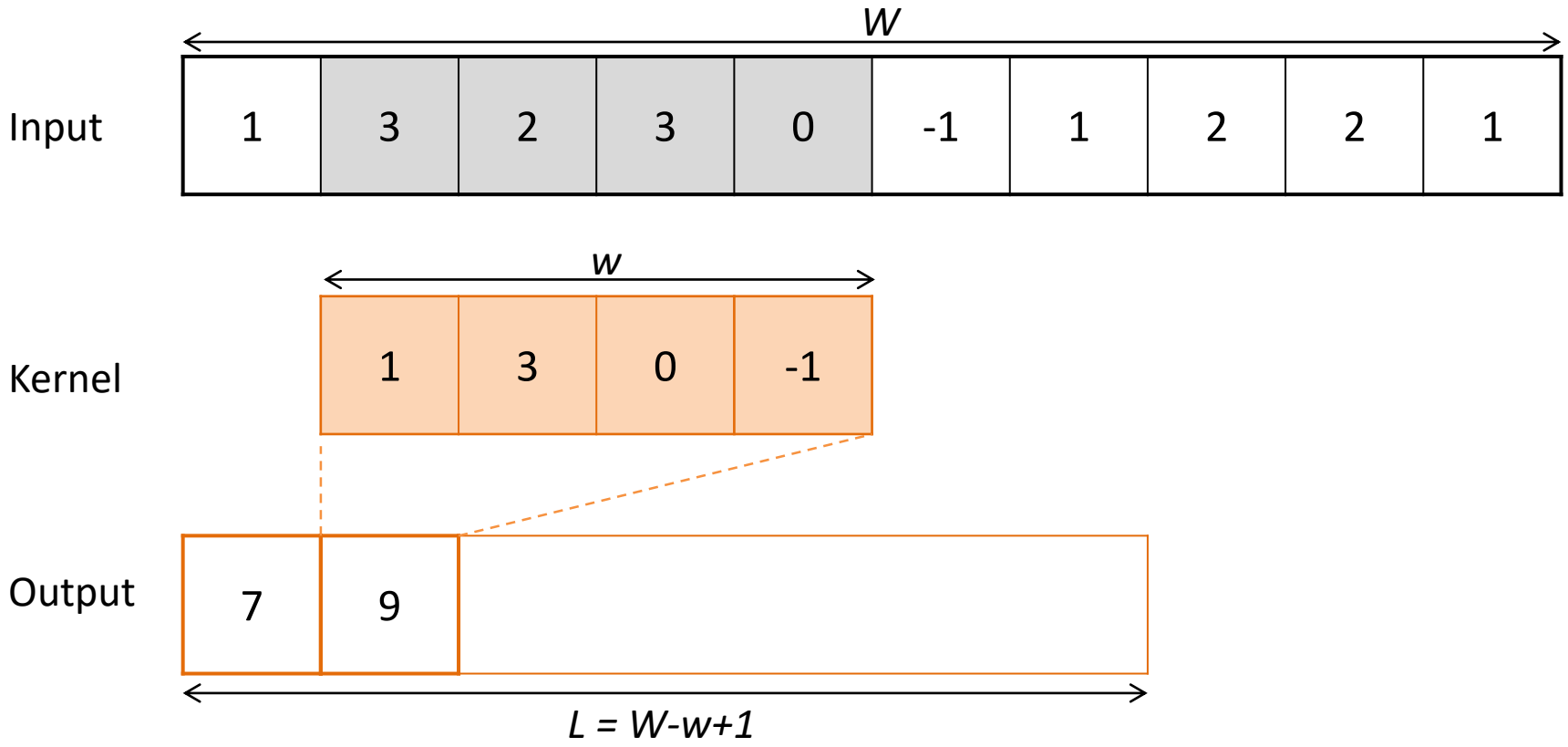
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

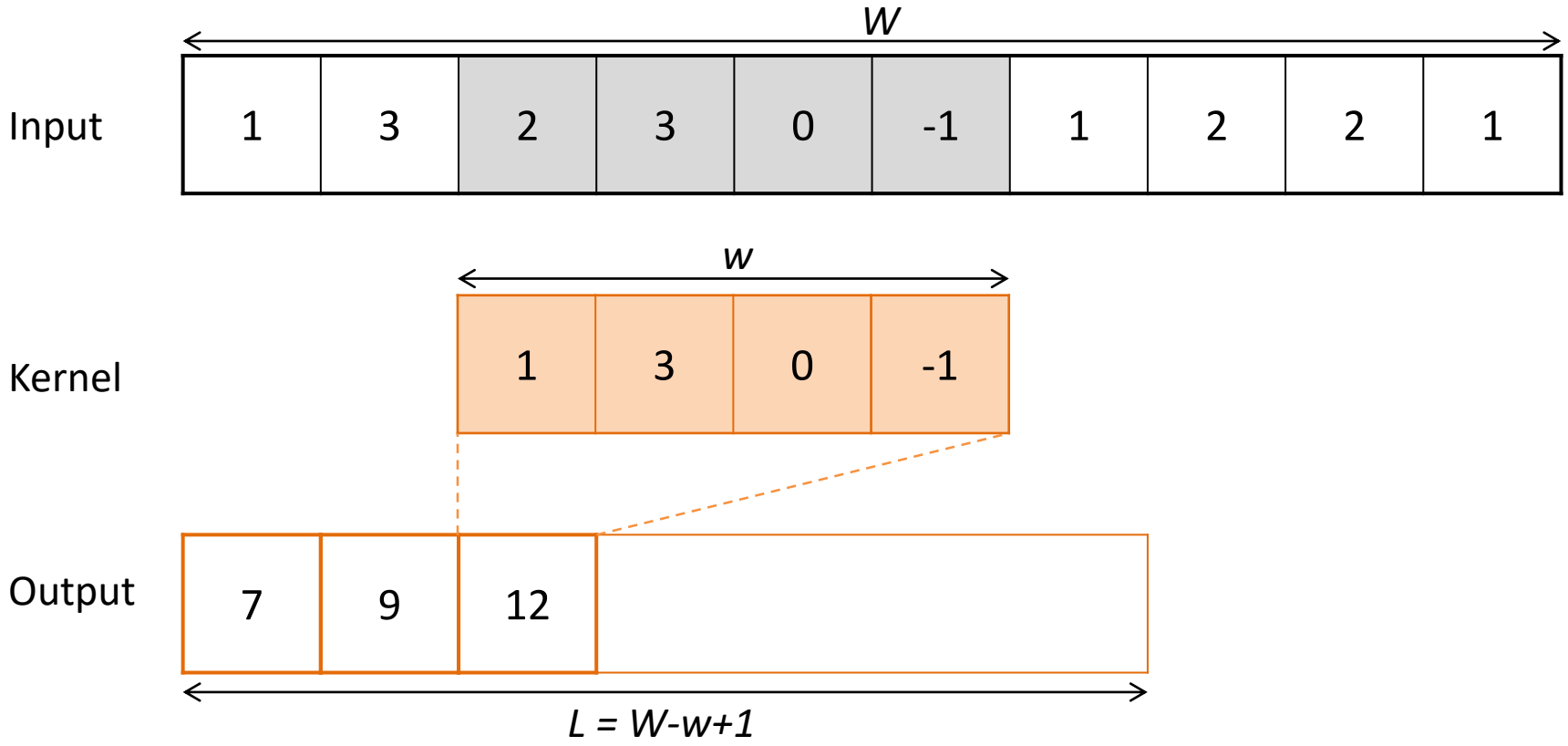
1D Convolution

- (actually cross-correlation)



1D Convolution

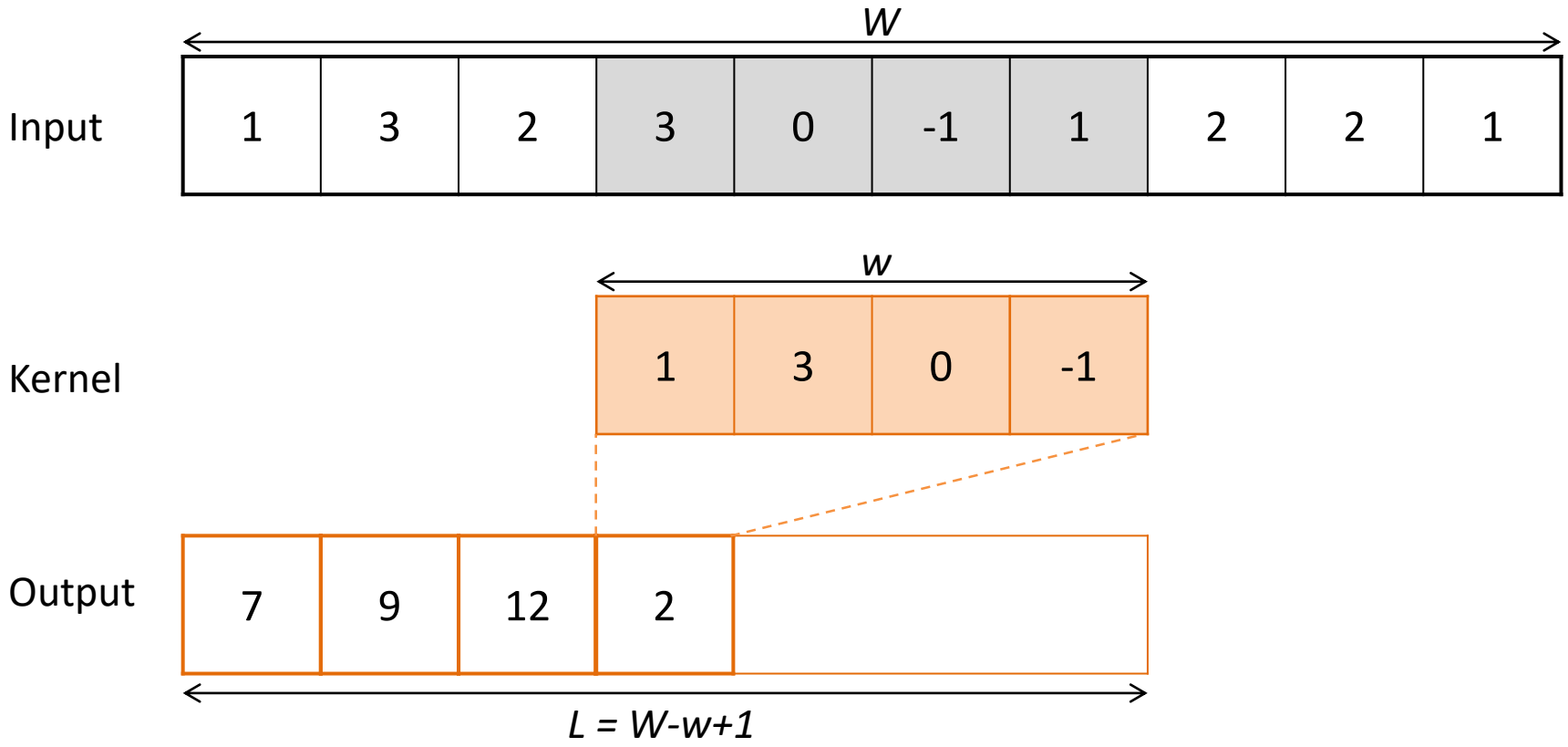
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

1D Convolution

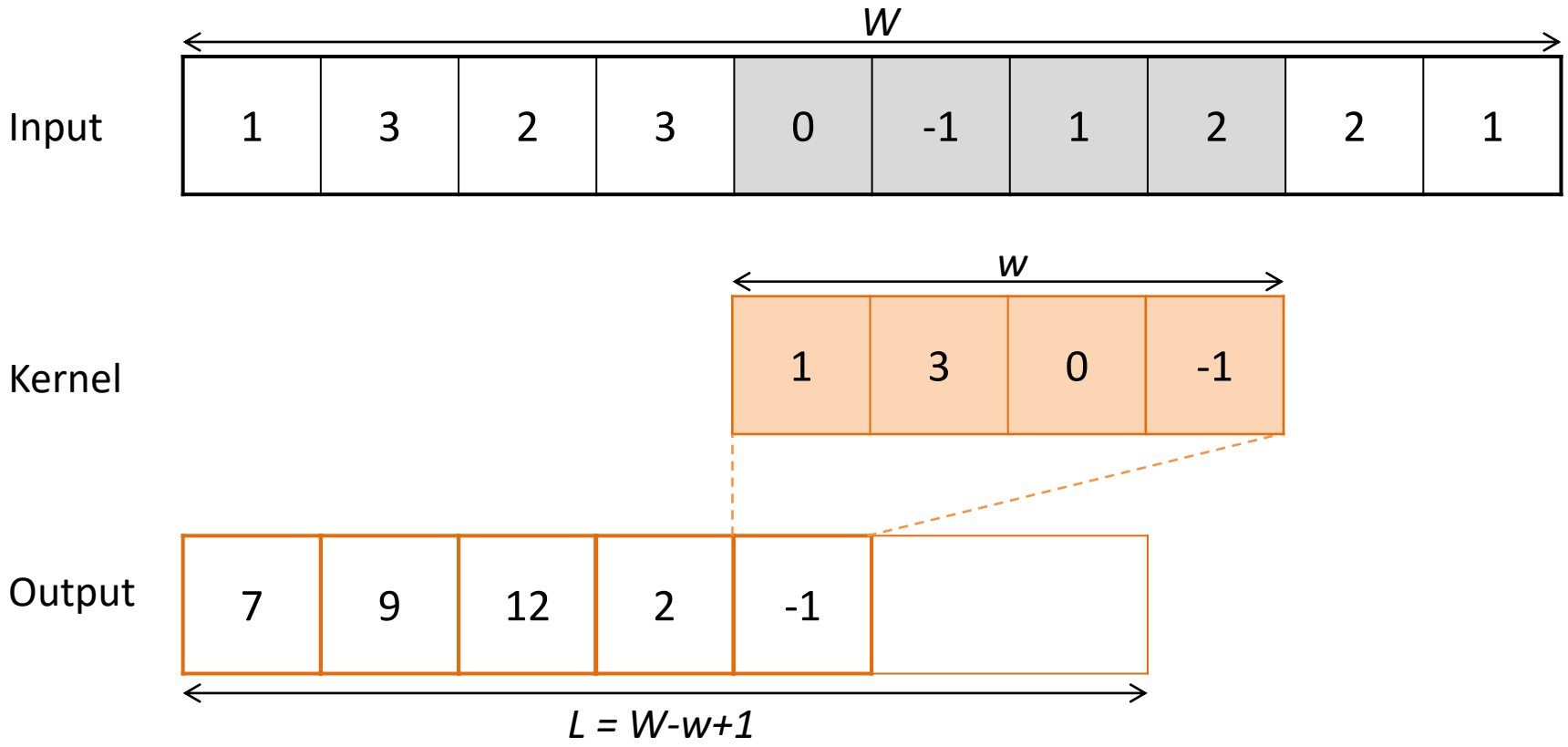
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

1D Convolution

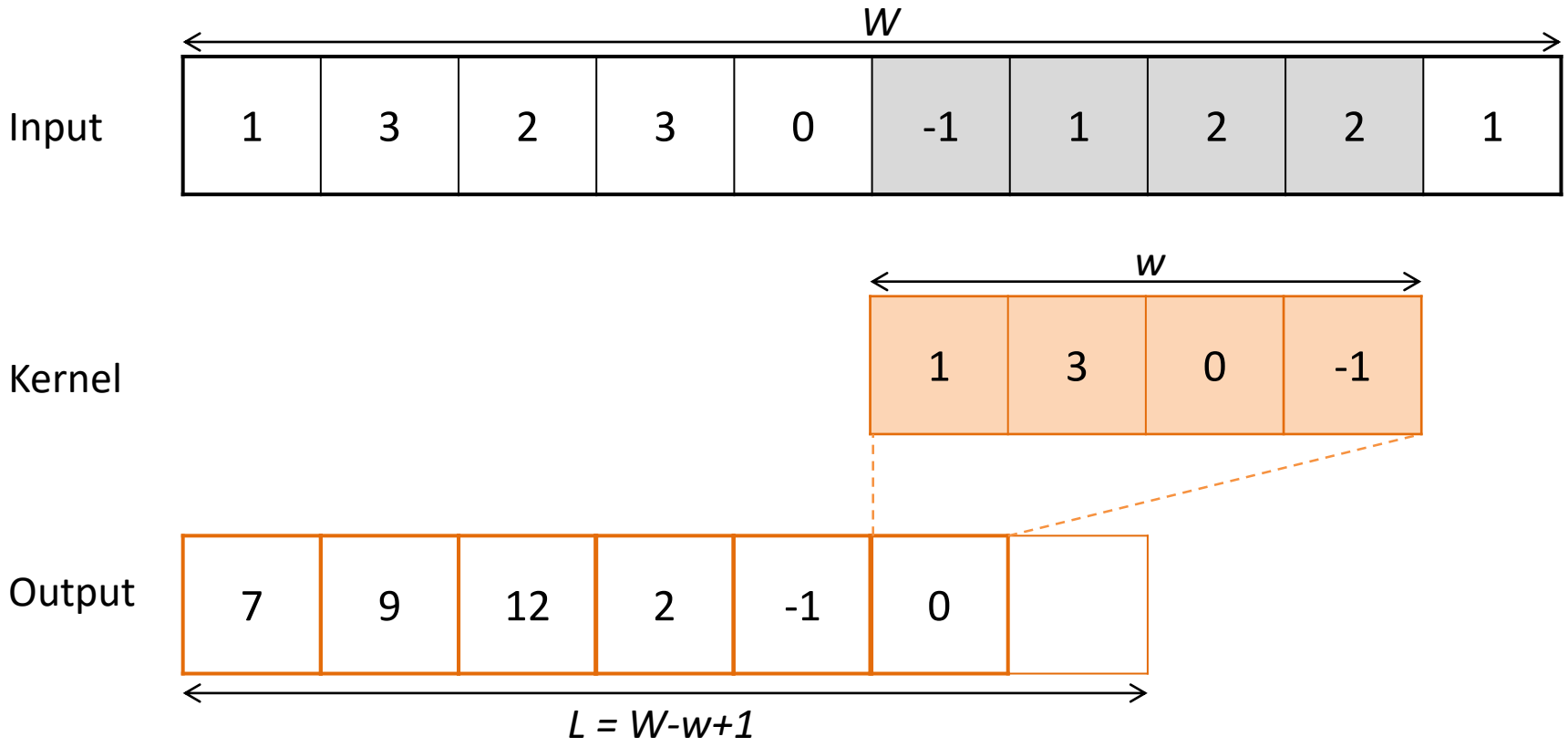
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

1D Convolution

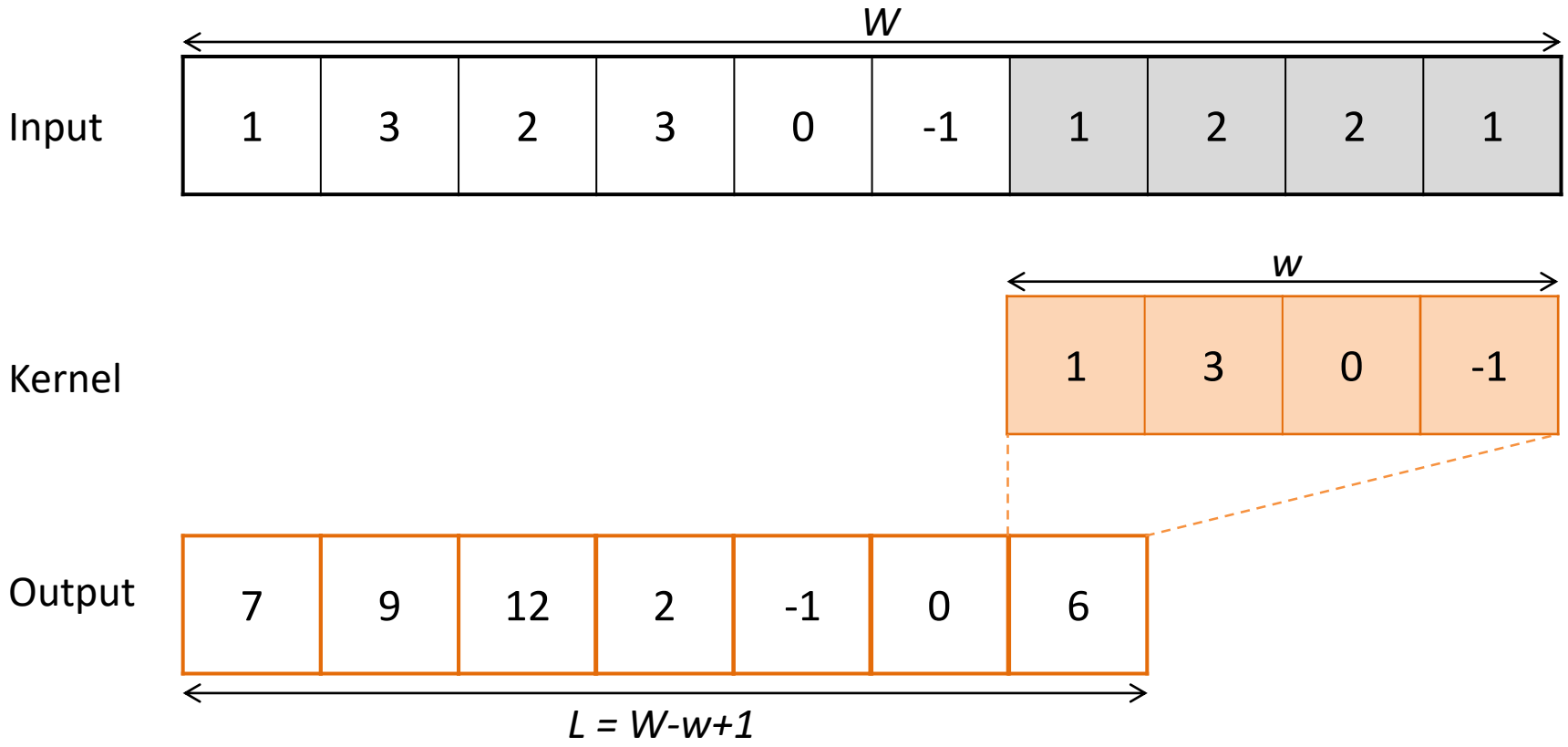
- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

1D Convolution

- (actually cross-correlation)



Source: Dr. Francois Fleuret at EPFL

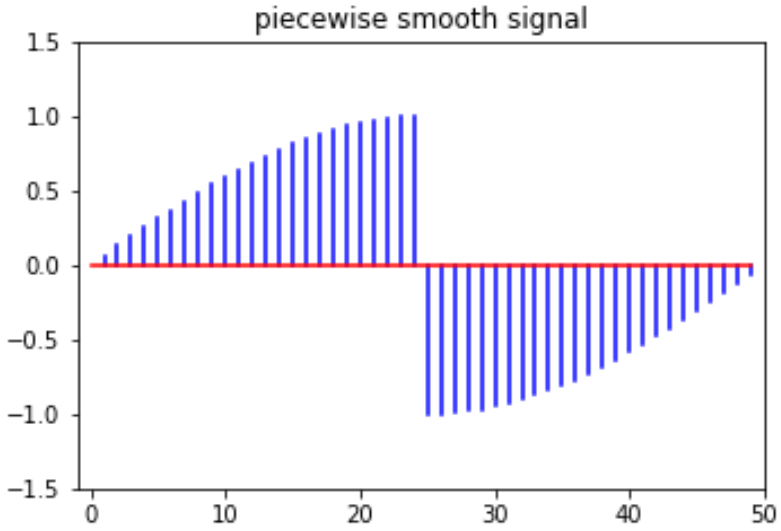
De-noising a Piecewise Smooth Signal

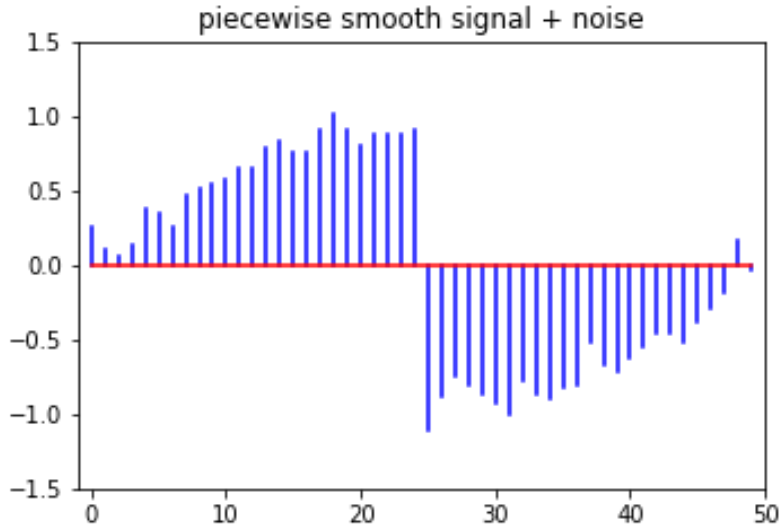
- Moving average (MA) filter
  - A moving average is the unweighted mean of the previous m data

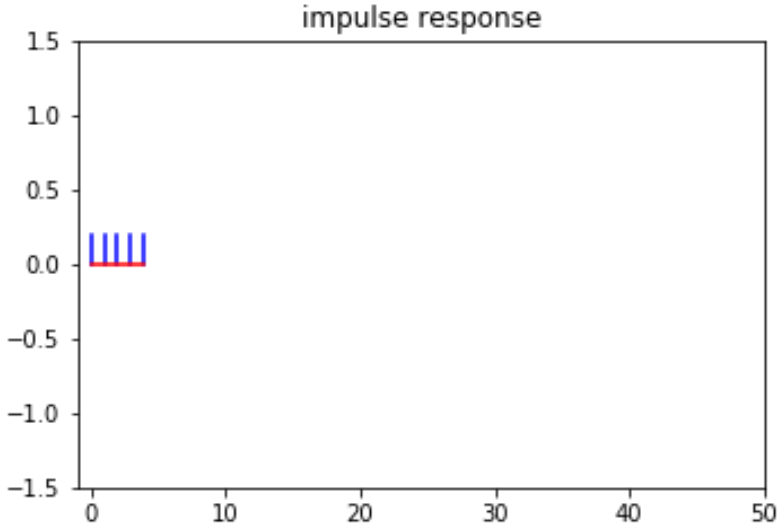
$$\begin{aligned}\bar{x}[n] &= \frac{x[n] + x[n-1] + \cdots + x[n-m+1]}{m} \\ &= (x[n], x[n-1], \cdots, x[n-m+1]) * \left(\frac{1}{m}, \frac{1}{m}, \cdots, \frac{1}{m}\right)\end{aligned}$$

- Convolution with $\left(\frac{1}{m}, \frac{1}{m}, \cdots, \frac{1}{m}\right)$

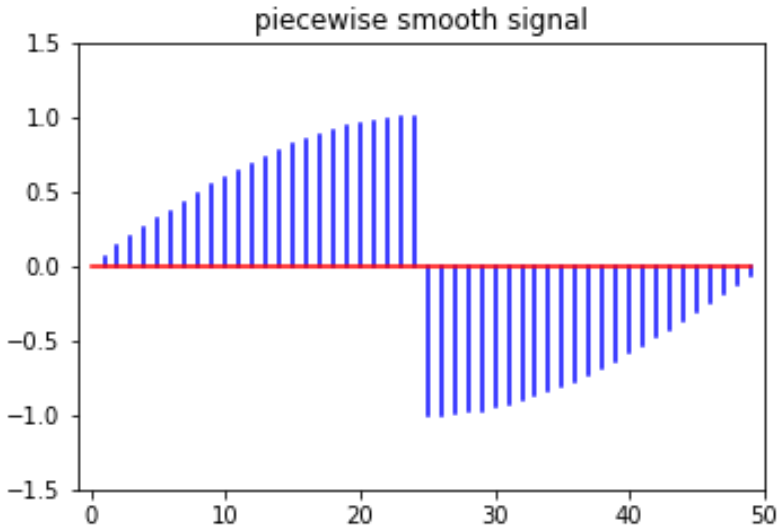
De-noising a Piecewise Smooth Signal

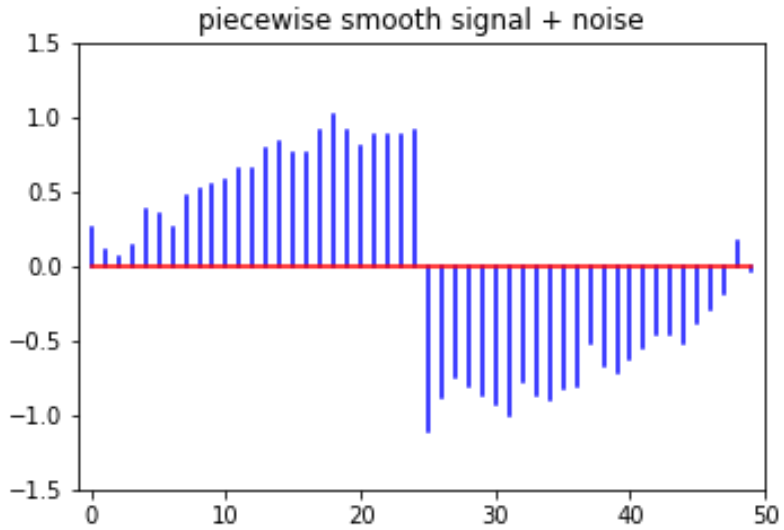


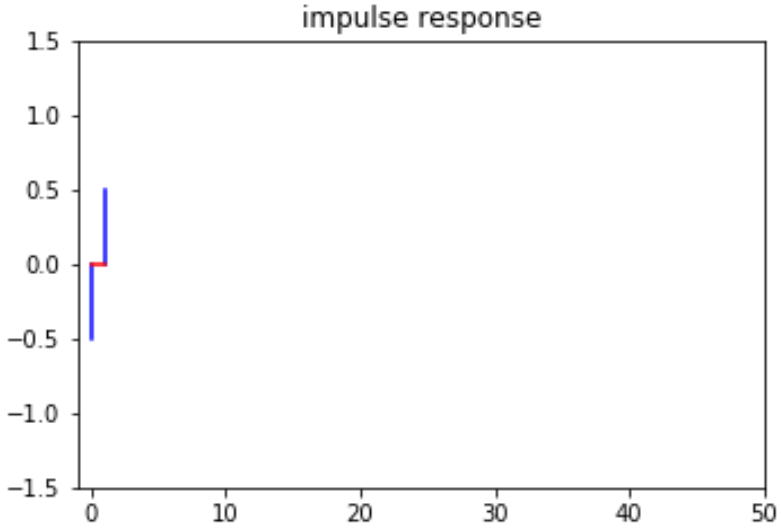




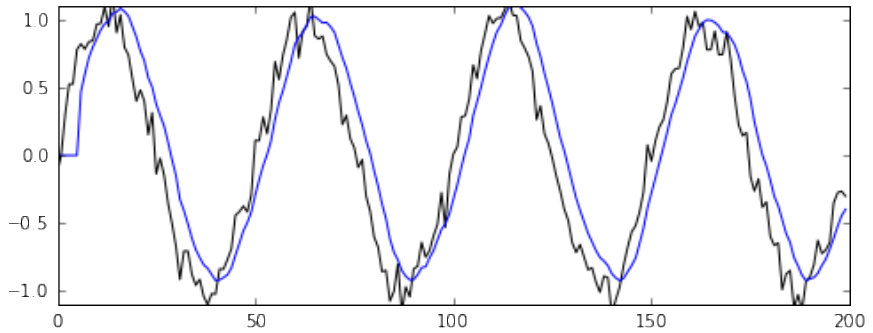
Edge Detection

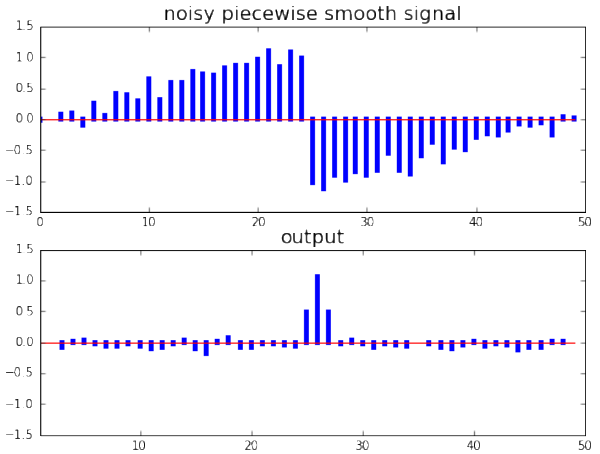


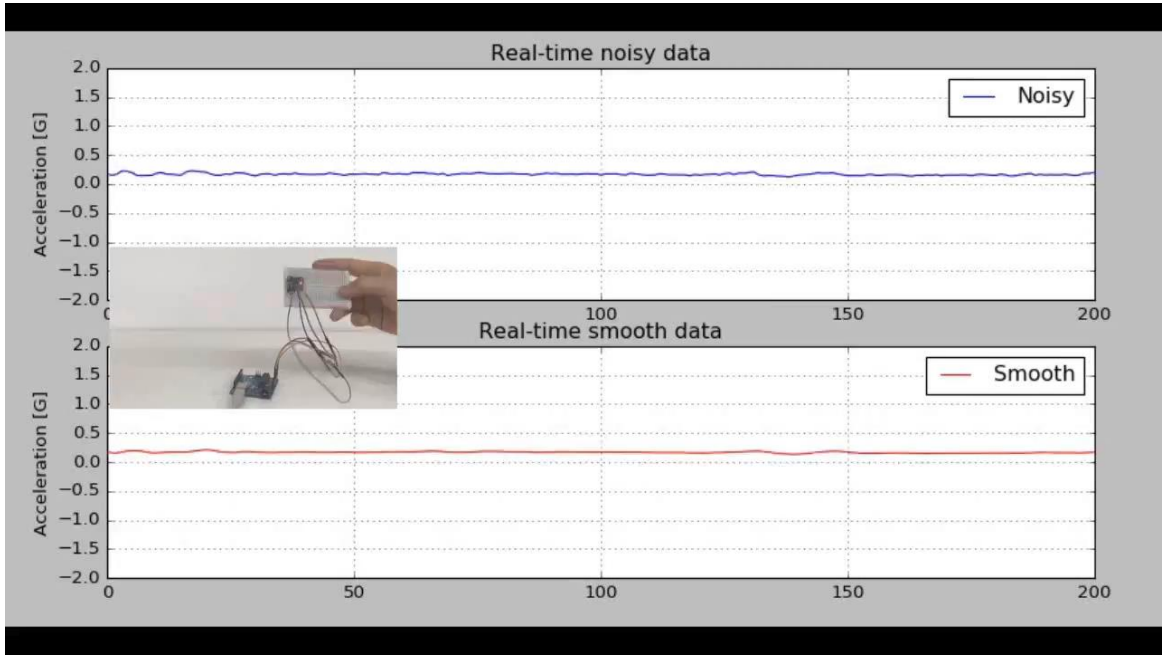


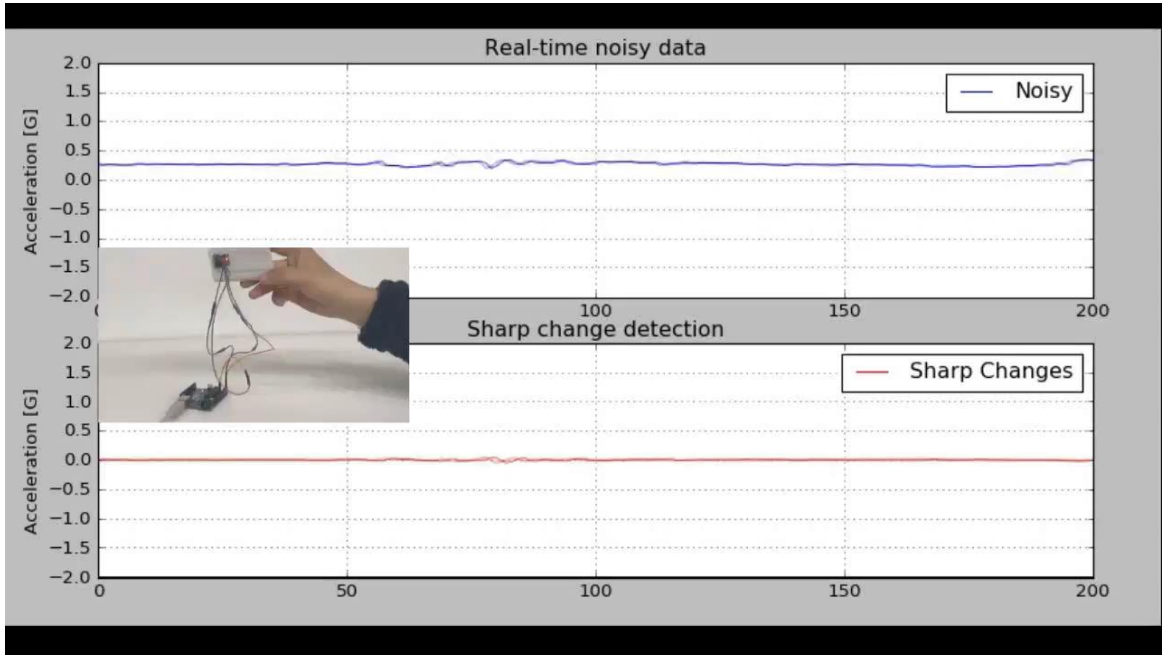


Smoothing and Detection of Abrupt Changes



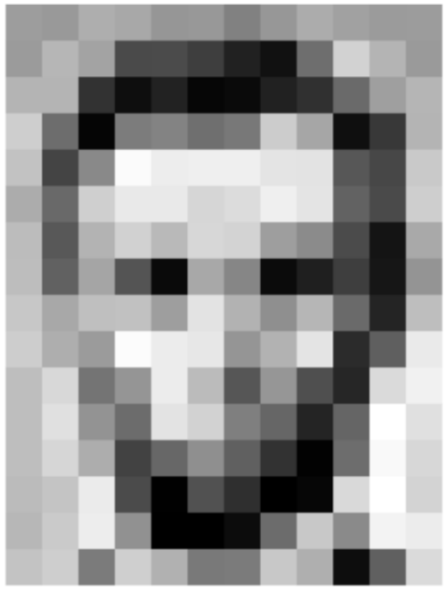






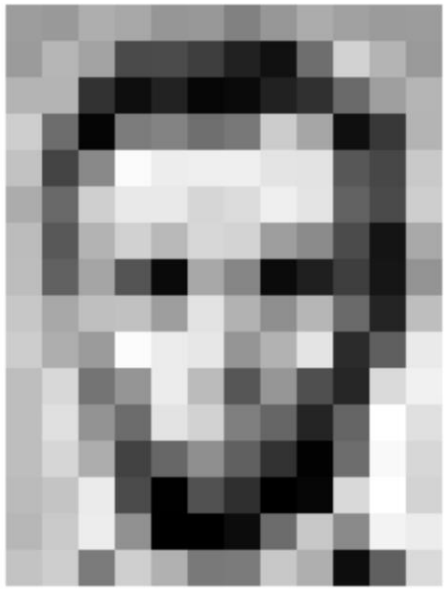
Images

Images Are Numbers



Source: 6.S191 Intro. to Deep Learning at MIT

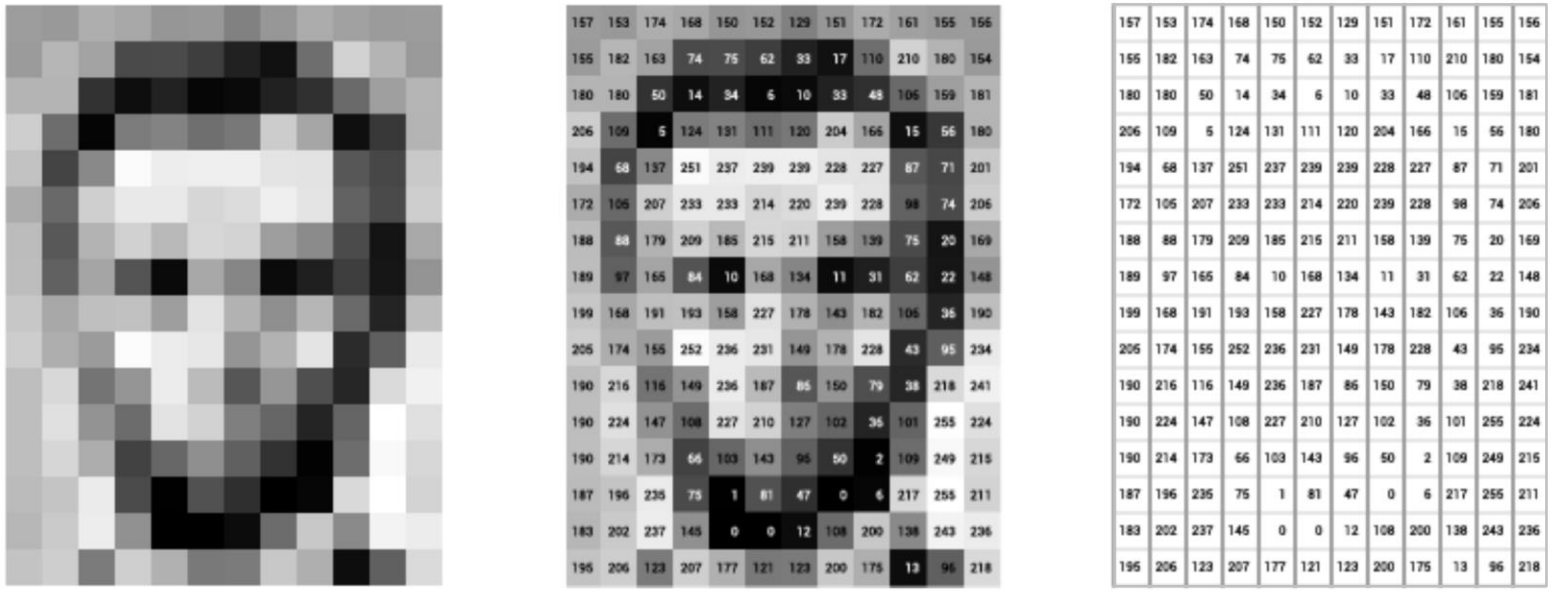
Images Are Numbers



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Source: 6.S191 Intro. to Deep Learning at MIT

Images Are Numbers



An image is just a matrix of numbers [0,255]!
i.e., 1080x1080x3 for an RGB image

Source: 6.S191 Intro. to Deep Learning at MIT

Images

Original image

R

G

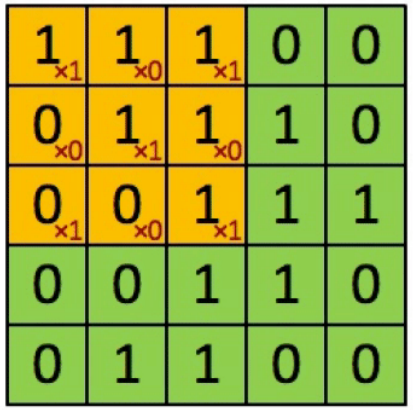
B

Gray image

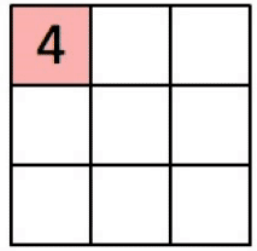
2D Convolution

Convolution on Image (= Convolution in 2D)

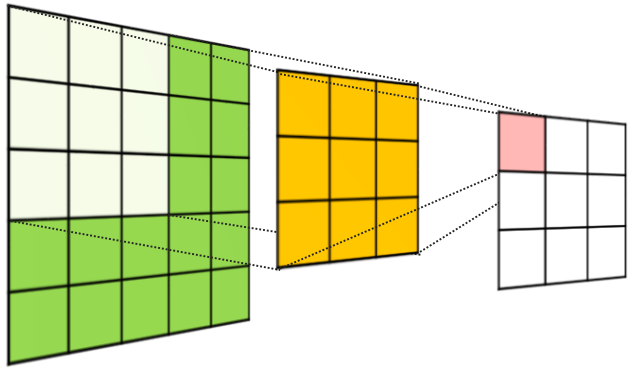
- Filter (or Kernel)
  - Discrete convolution can be viewed as **element-wise multiplication** by a matrix
  - Modify or enhance an image by filtering
  - Filter images to emphasize certain features or remove other features
  - Filtering includes smoothing, sharpening and edge enhancement



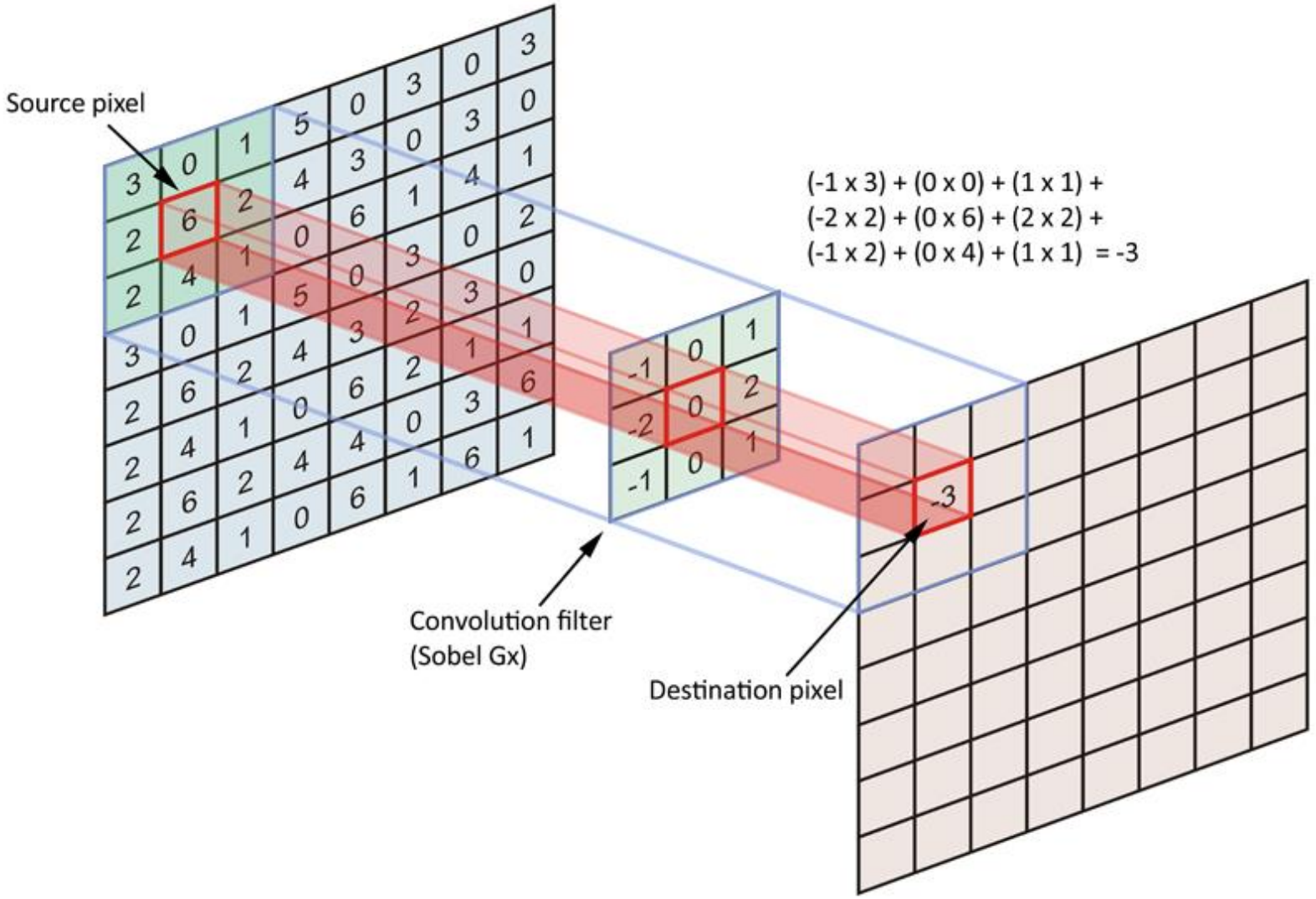
Image



Convolved Feature



Convolution on Image (= Convolution in 2D)



Convolution on Image

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Image

Convolution on Image

```
M = np.ones([3,3])/9

img_conv = signal.convolve2d(img, M, 'same')
```

Noisy Image

Smoothed Image

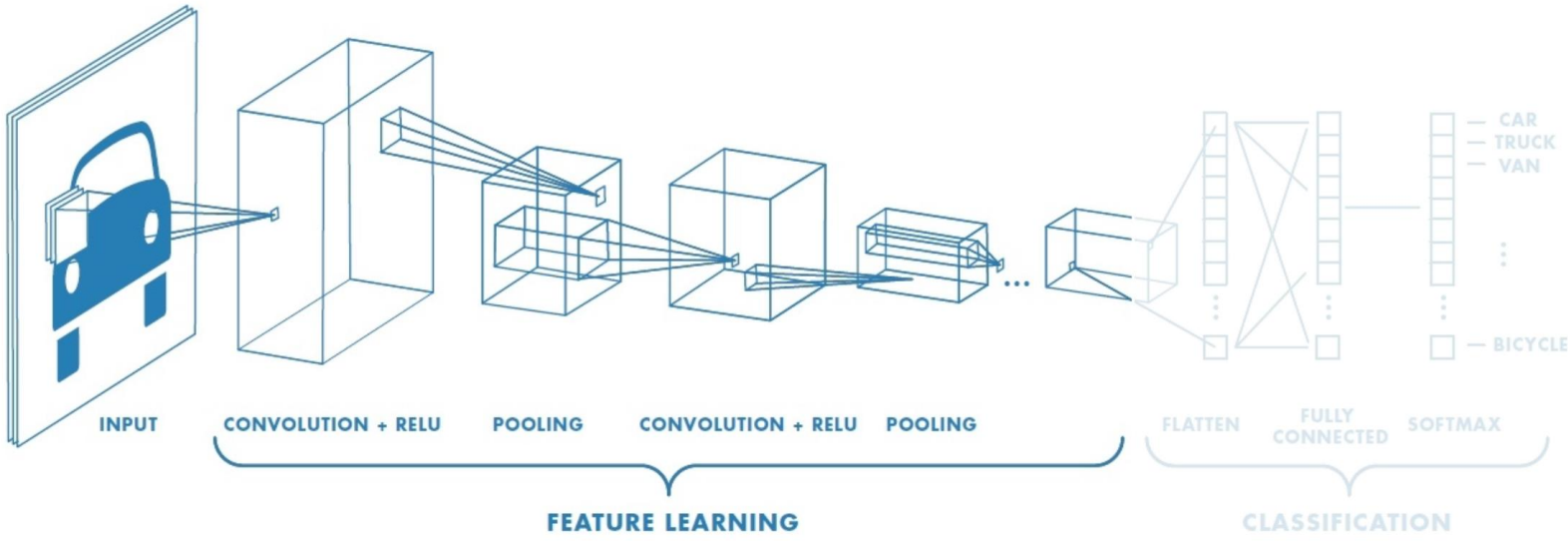
How to Find the Right Kernels

- We learn many different kernels that make specific effect on images
- Let’s apply an opposite approach
- We are not designing the kernel, but are learning the kernel from data
- Can learn feature extractor from data using a deep learning framework

Convolutional Neural Networks (CNN)

CNNs for Classification: Feature Learning

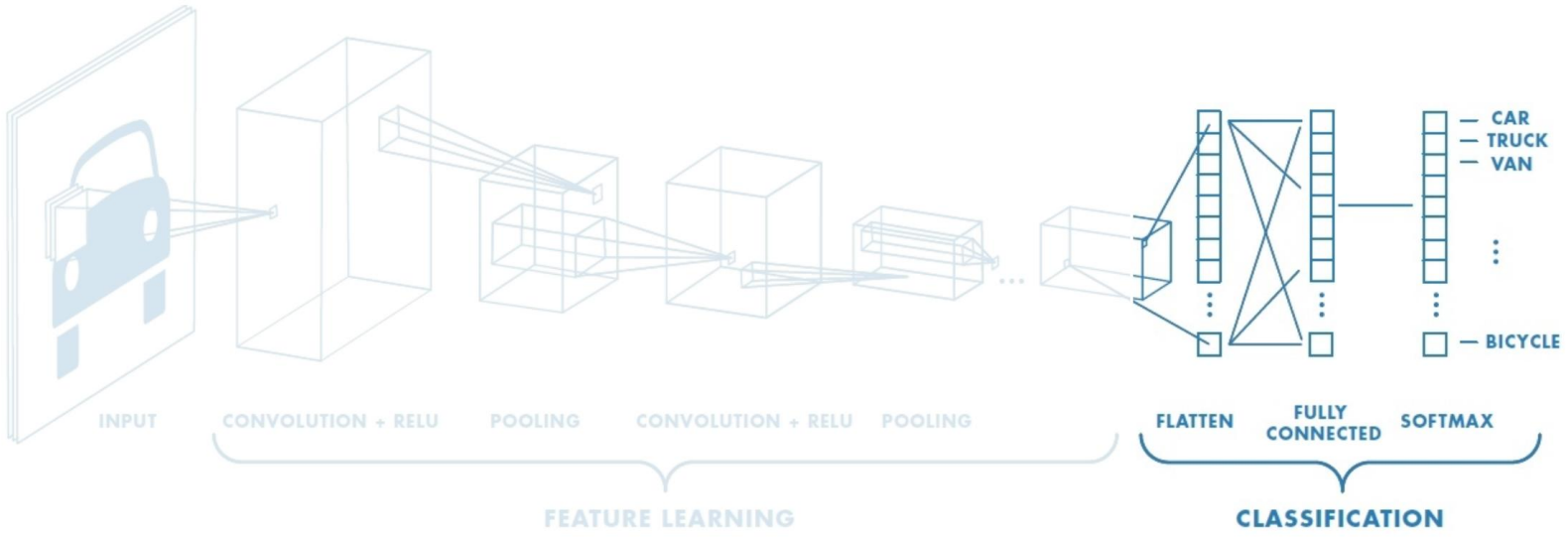
- Learn features in input image through convolution
- Introduce non-linearity through activation function (real-world data is non-linear!)
- Reduce dimensionality and preserve spatial invariance with pooling



Source: 6.S191 Intro. to Deep Learning at MIT

CNNs for Classification: Class Probabilities

- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as probability of image belonging to a particular class

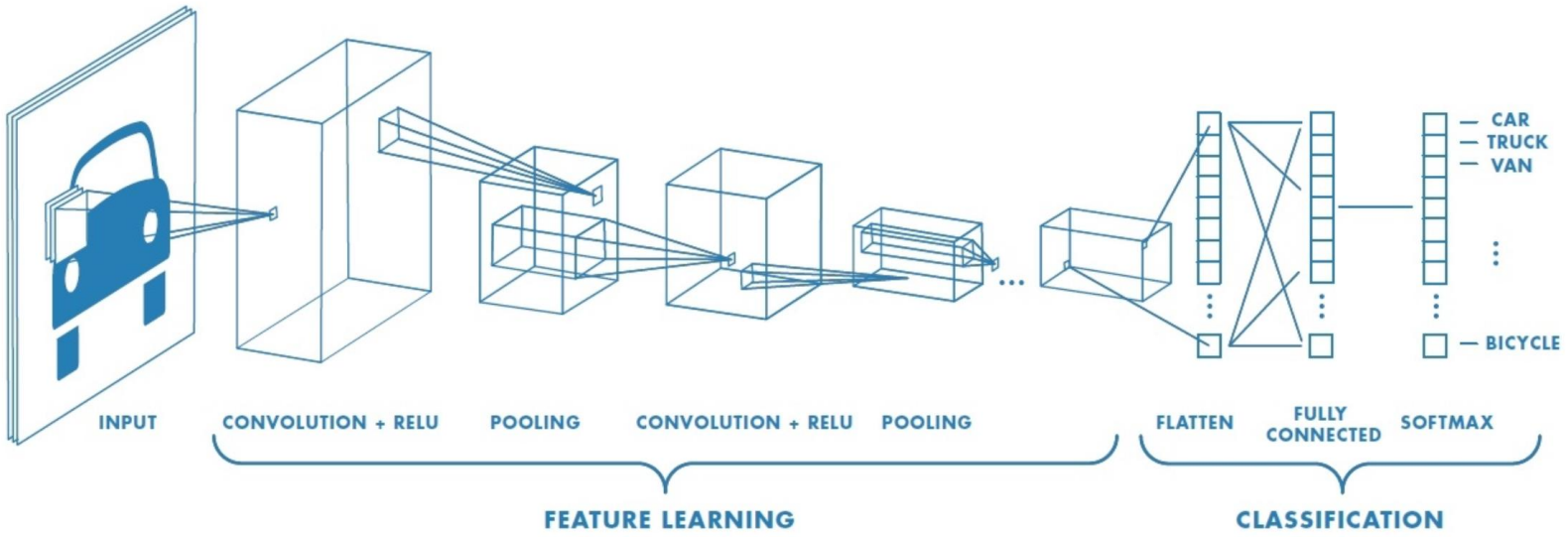


$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Source: 6.S191 Intro. to Deep Learning at MIT

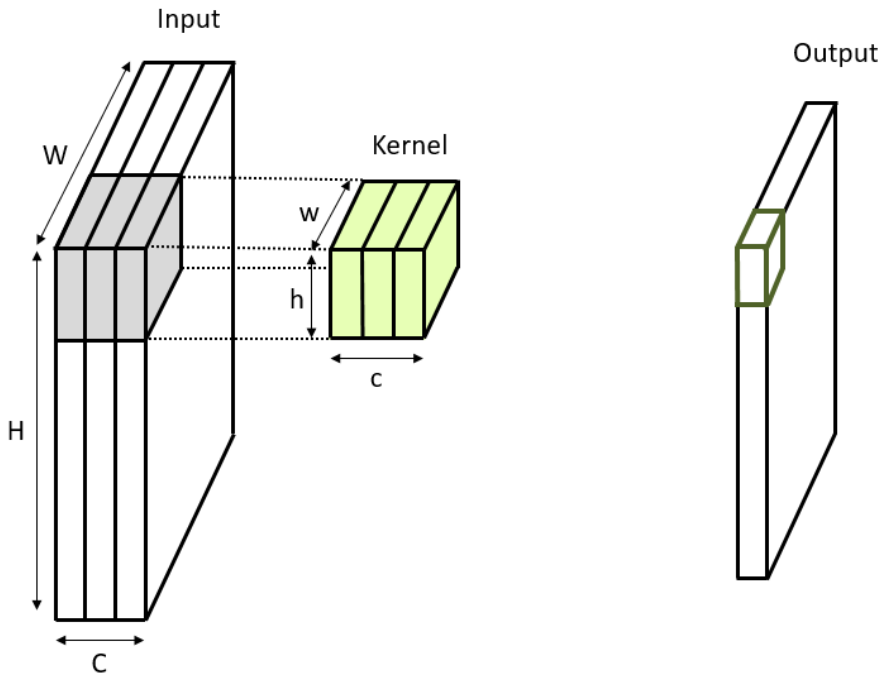
CNNs: Training with Backpropagation

- Learn weights for convolutional filters and fully connected layers
- Backpropagation: cross-entropy loss



Source: 6.S191 Intro. to Deep Learning at MIT

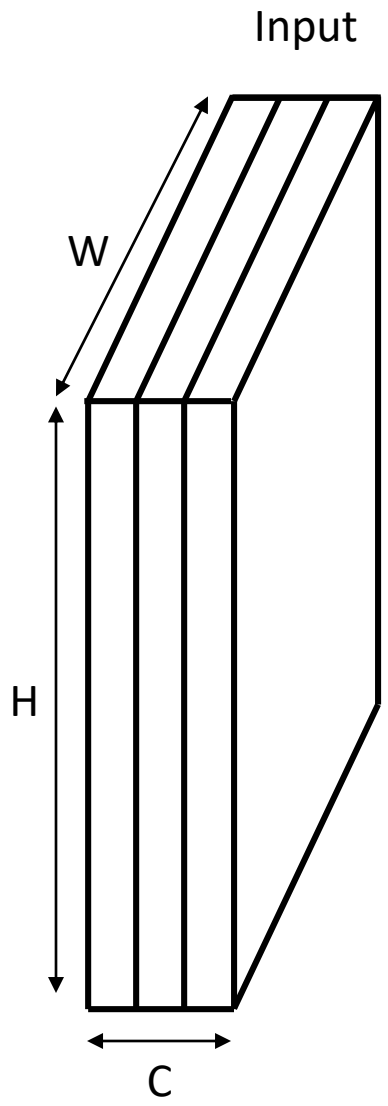
Channels



- Colored image = tensor of shape (height, width, channels)
- Convolutions are usually computed for each channel and summed:
- Kernel size aka receptive field (usually 1, 3, 5, 7, 11)

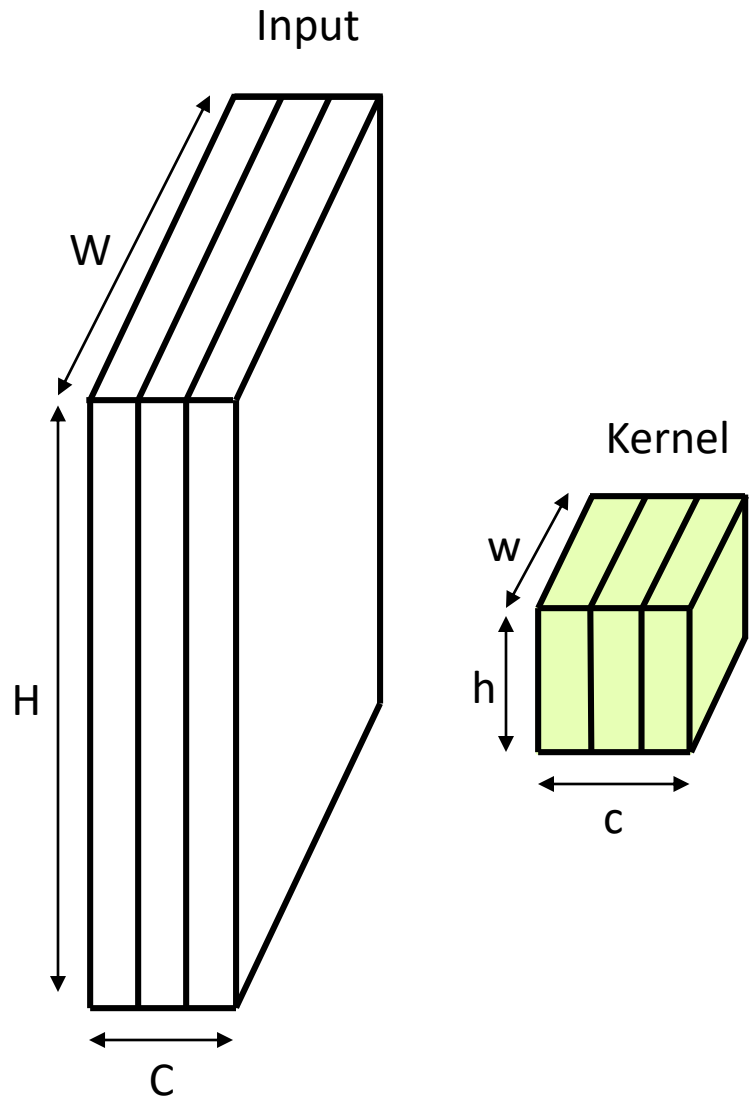
Source: Dr. Francois Fleuret at EPFL

Multi-channel 2D Convolution



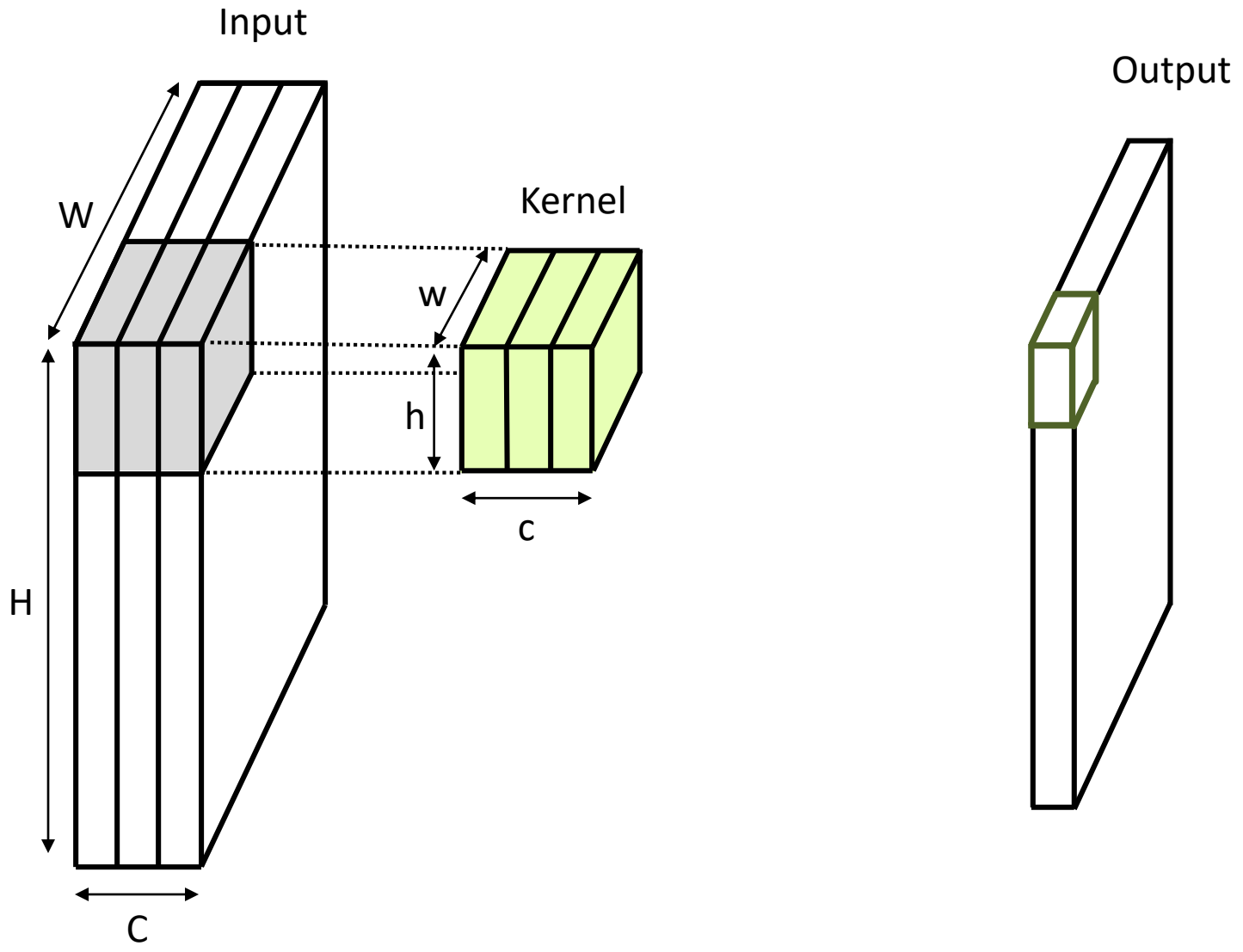
Source: Dr. Francois Fleuret at EPFL

Multi-channel 2D Convolution

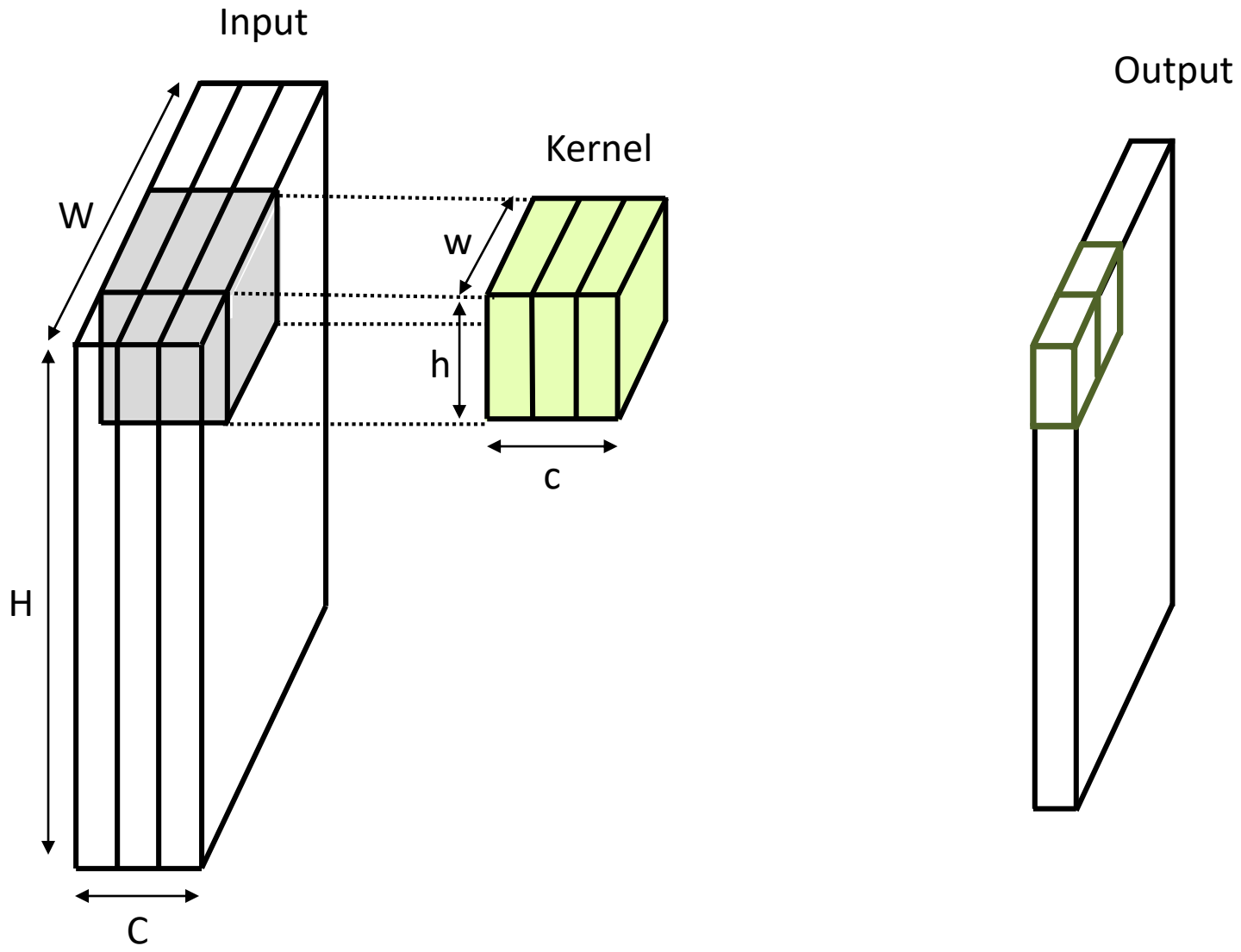


Source: Dr. Francois Fleuret at EPFL

Multi-channel 2D Convolution

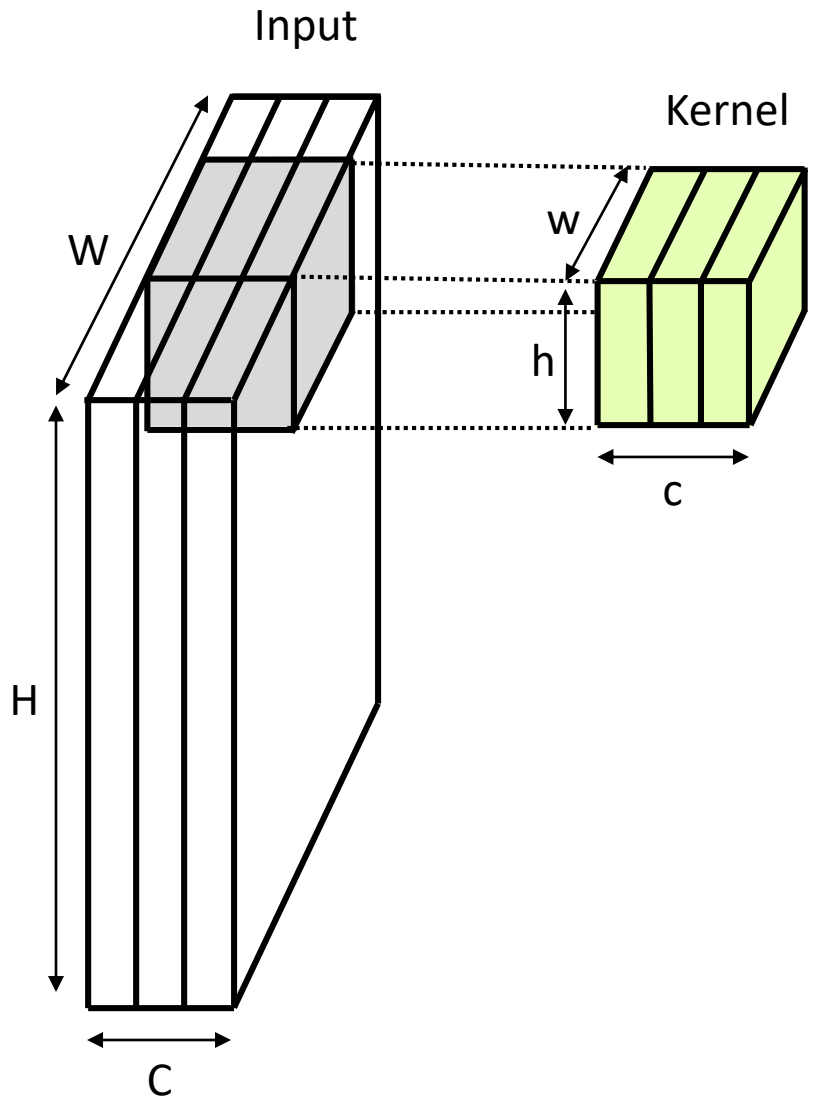


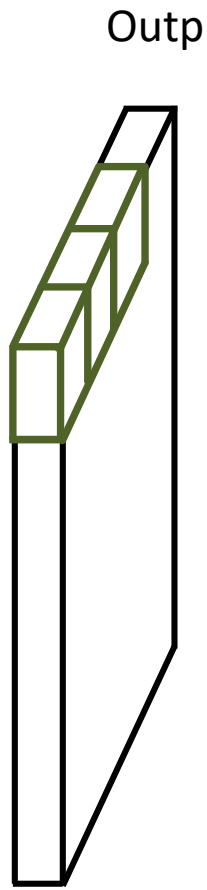
Multi-channel 2D Convolution



Source: Dr. Francois Fleuret at EPFL

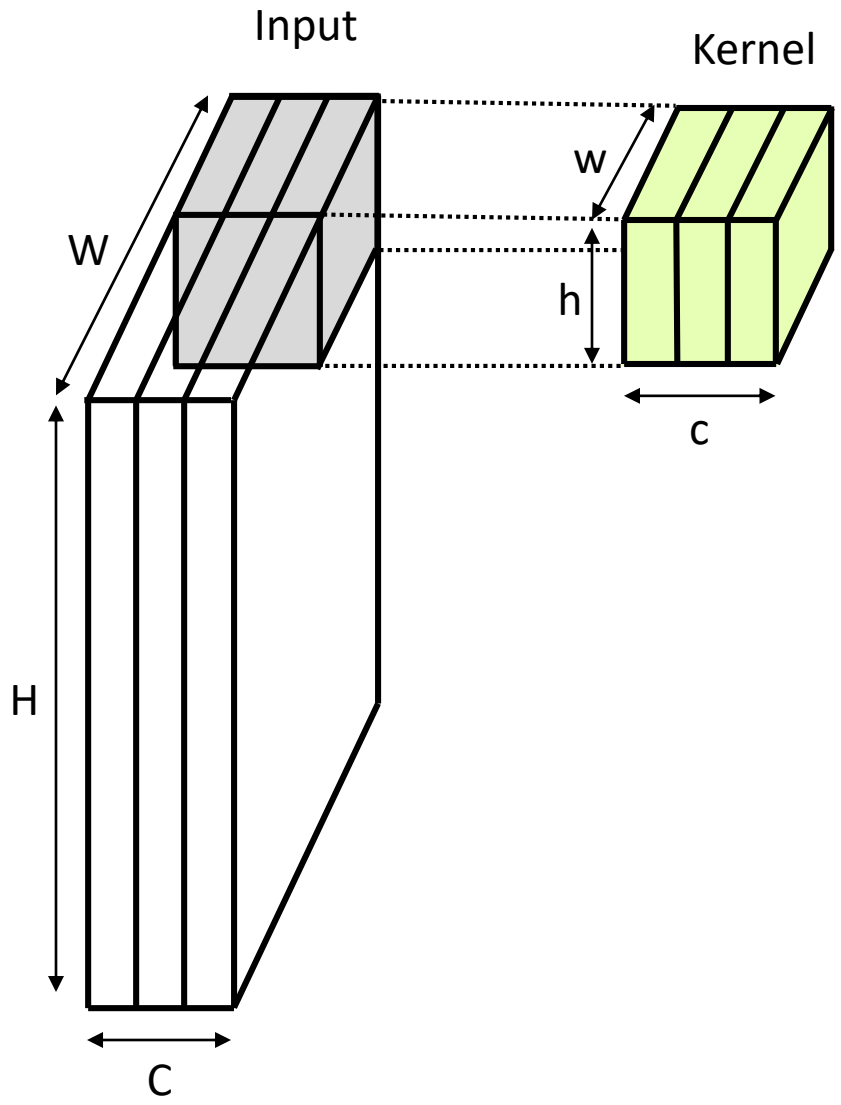
Multi-channel 2D Convolution

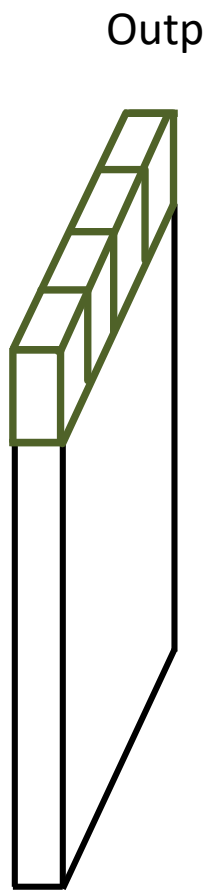




Source: Dr. Francois Fleuret at EPFL

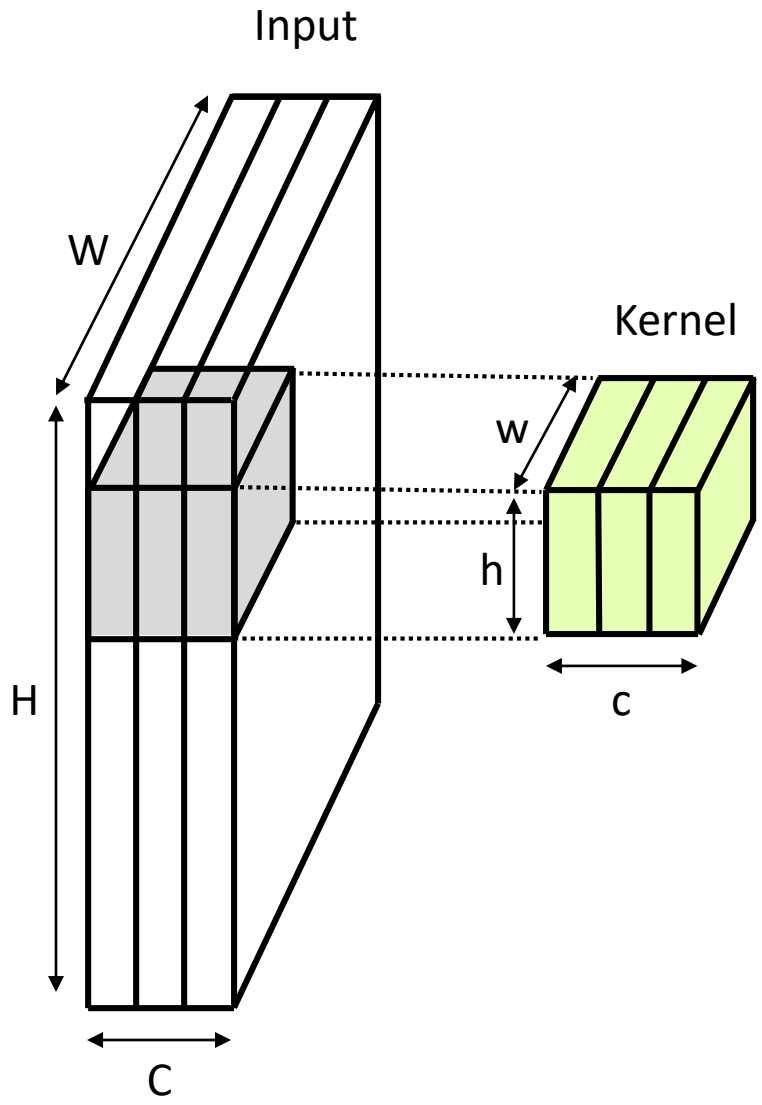
Multi-channel 2D Convolution

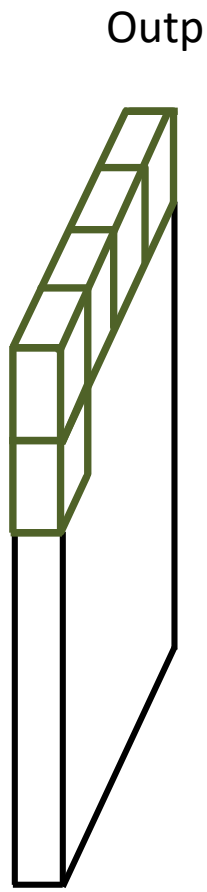




Source: Dr. Francois Fleuret at EPFL

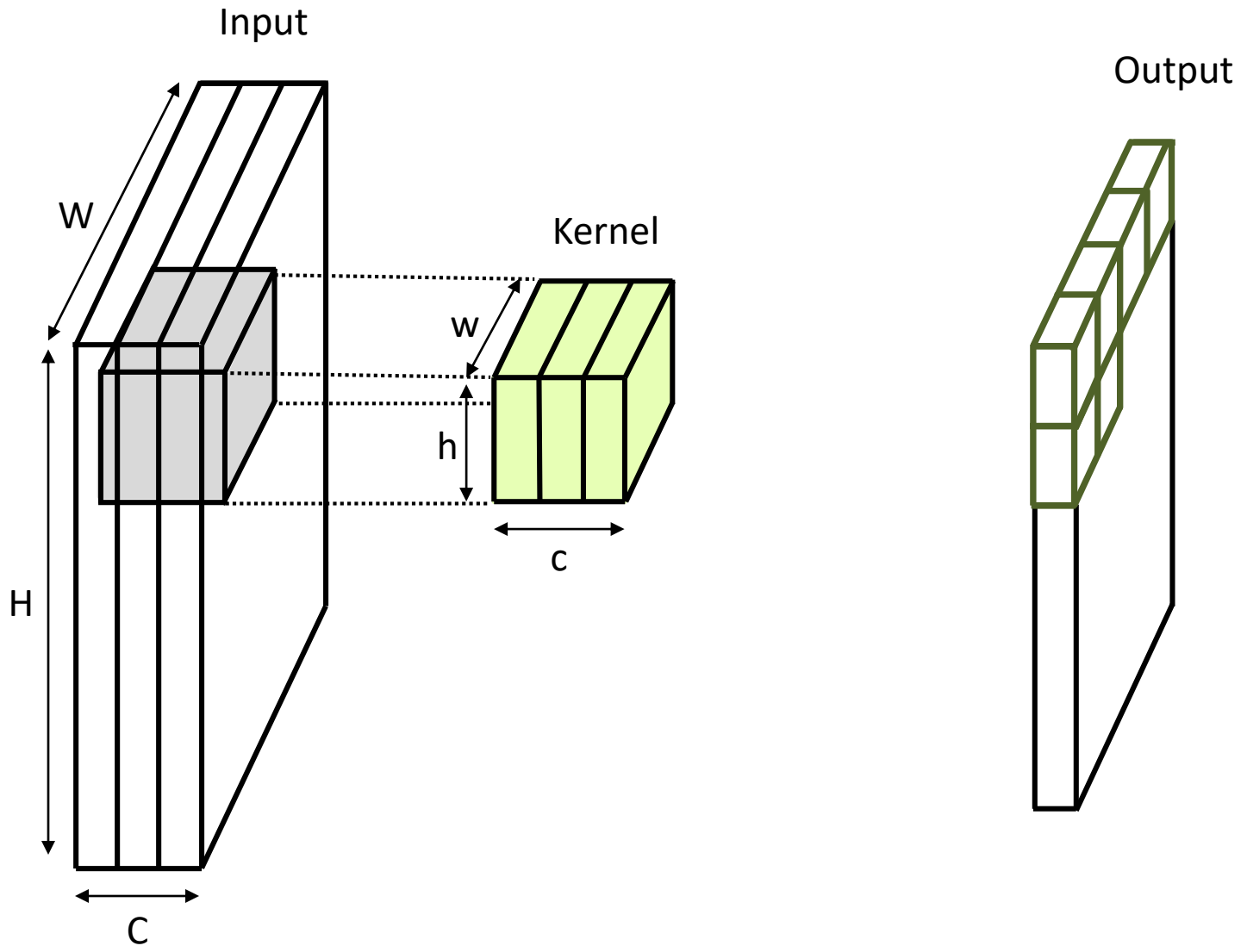
Multi-channel 2D Convolution





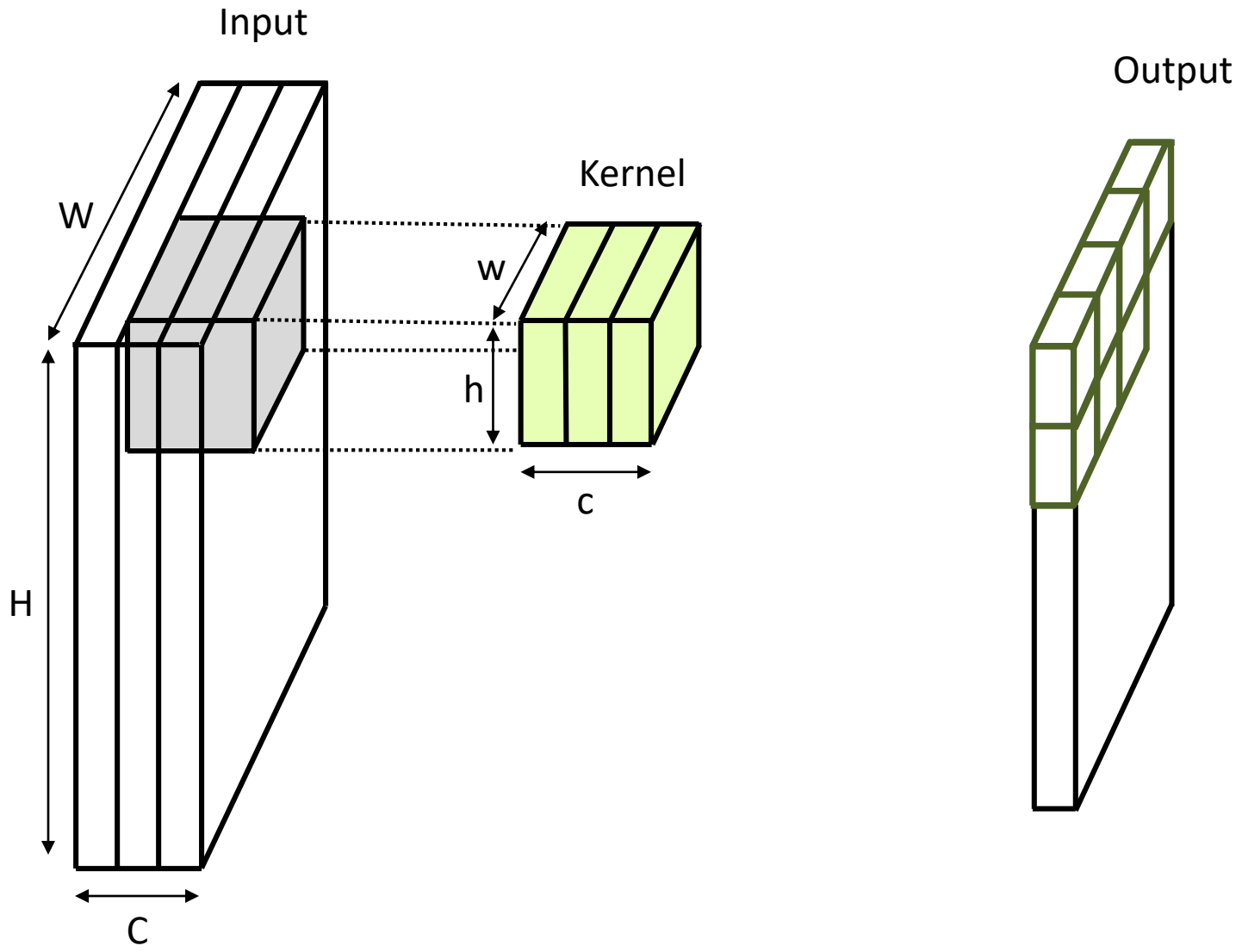
Source: Dr. Francois Fleuret at EPFL

Multi-channel 2D Convolution



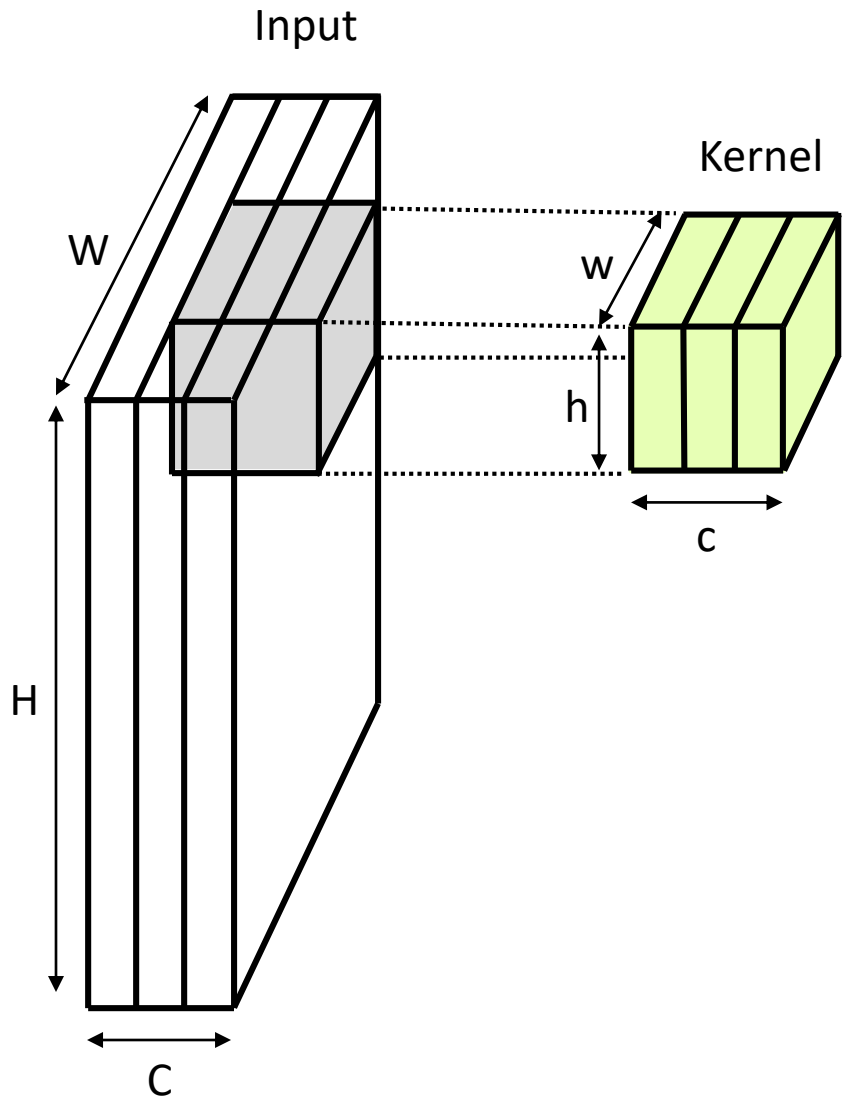
Source: Dr. Francois Fleuret at EPFL

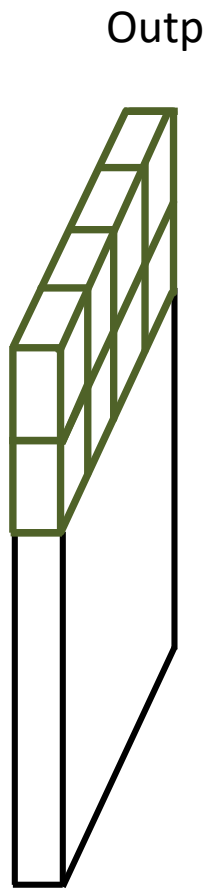
Multi-channel 2D Convolution



Source: Dr. Francois Fleuret at EPFL

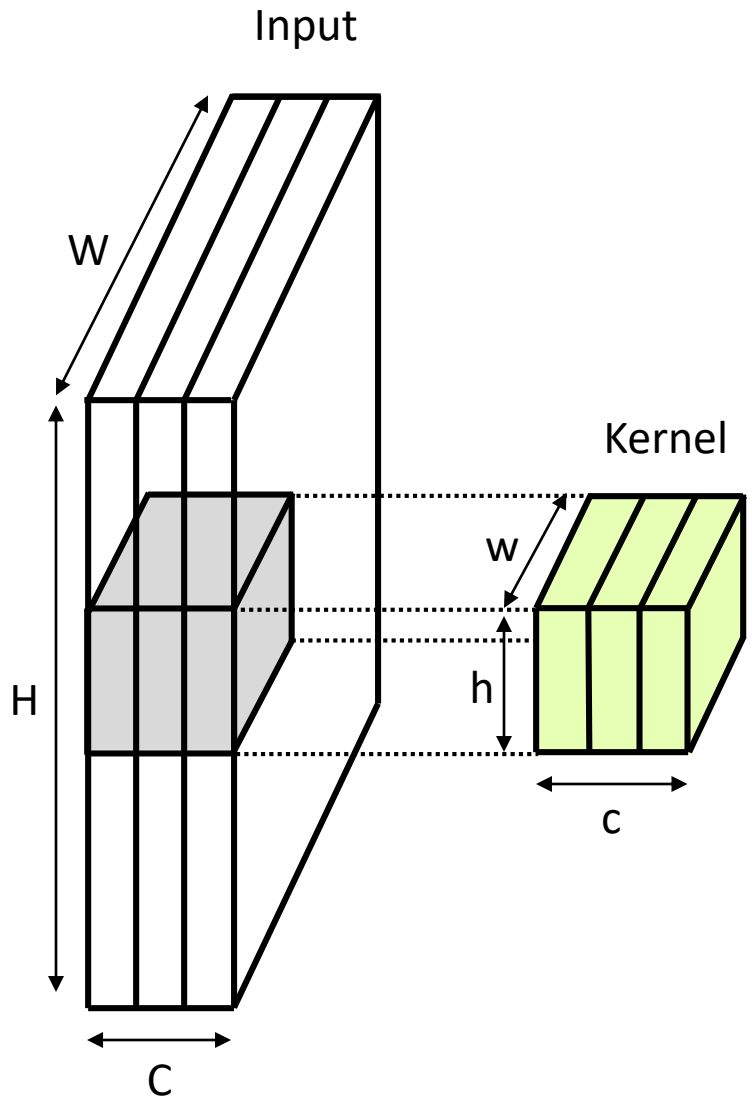
Multi-channel 2D Convolution

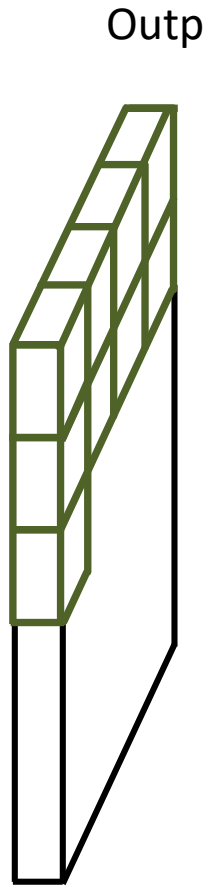




Source: Dr. Francois Fleuret at EPFL

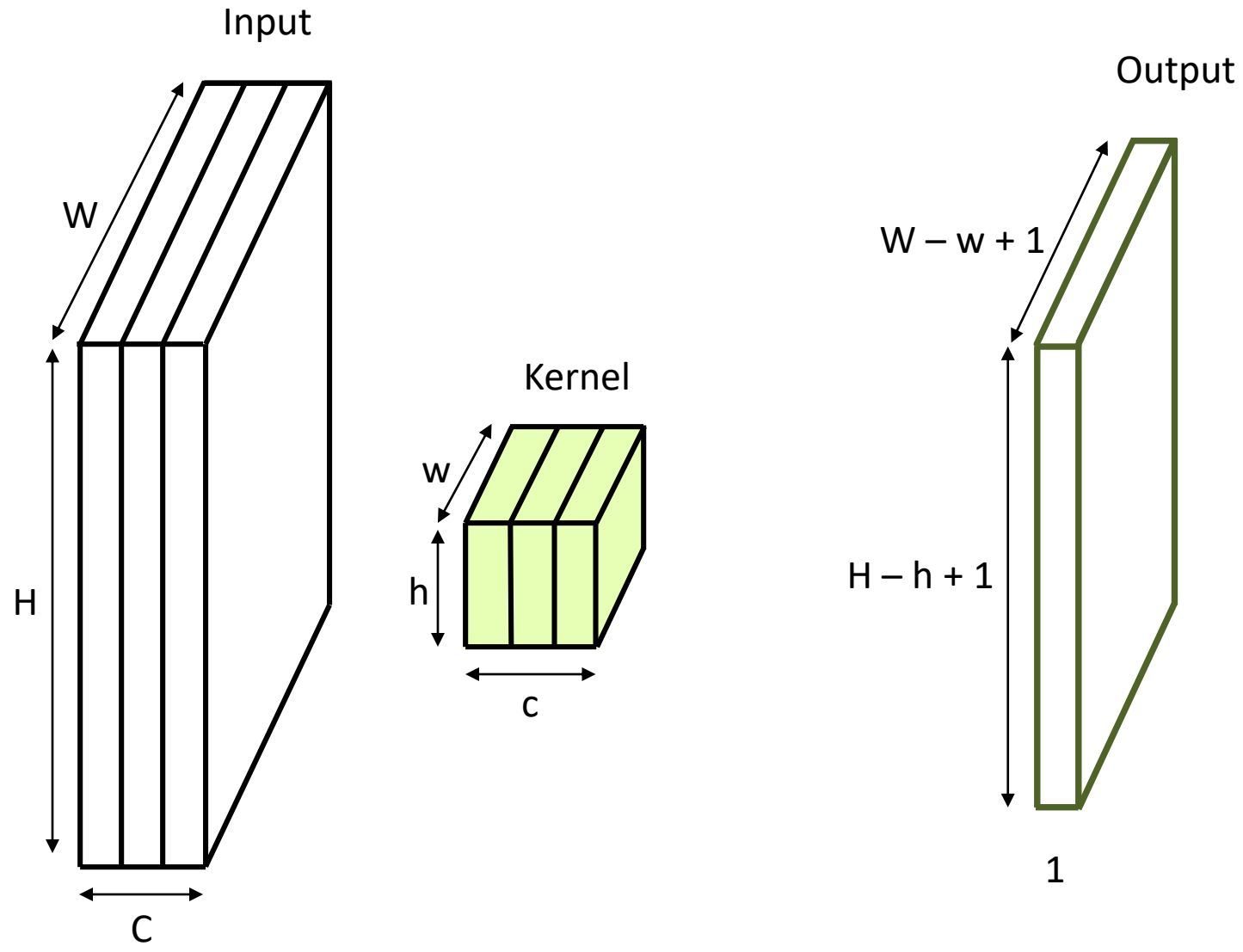
Multi-channel 2D Convolution





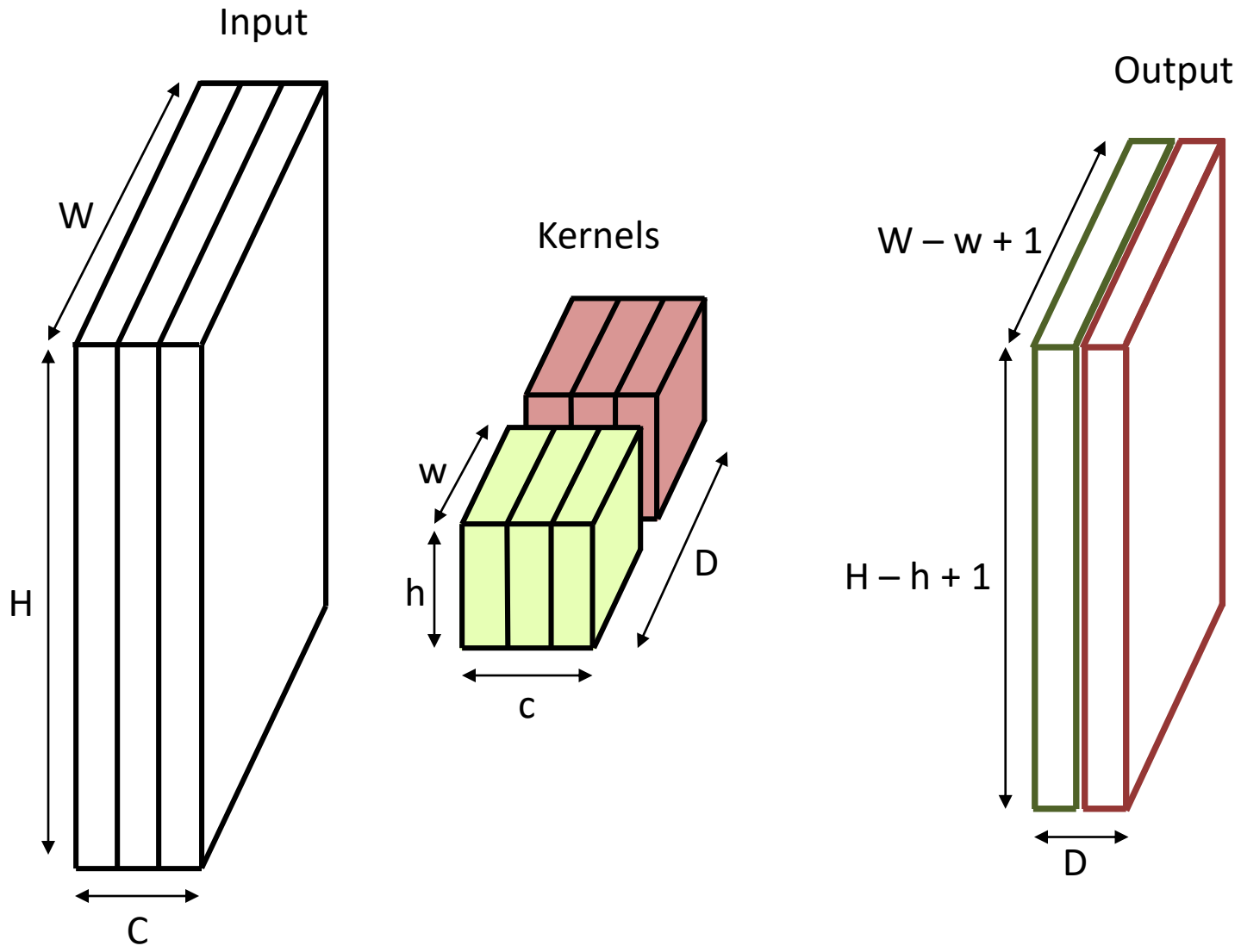
Source: Dr. Francois Fleuret at EPFL

Multi-channel 2D Convolution



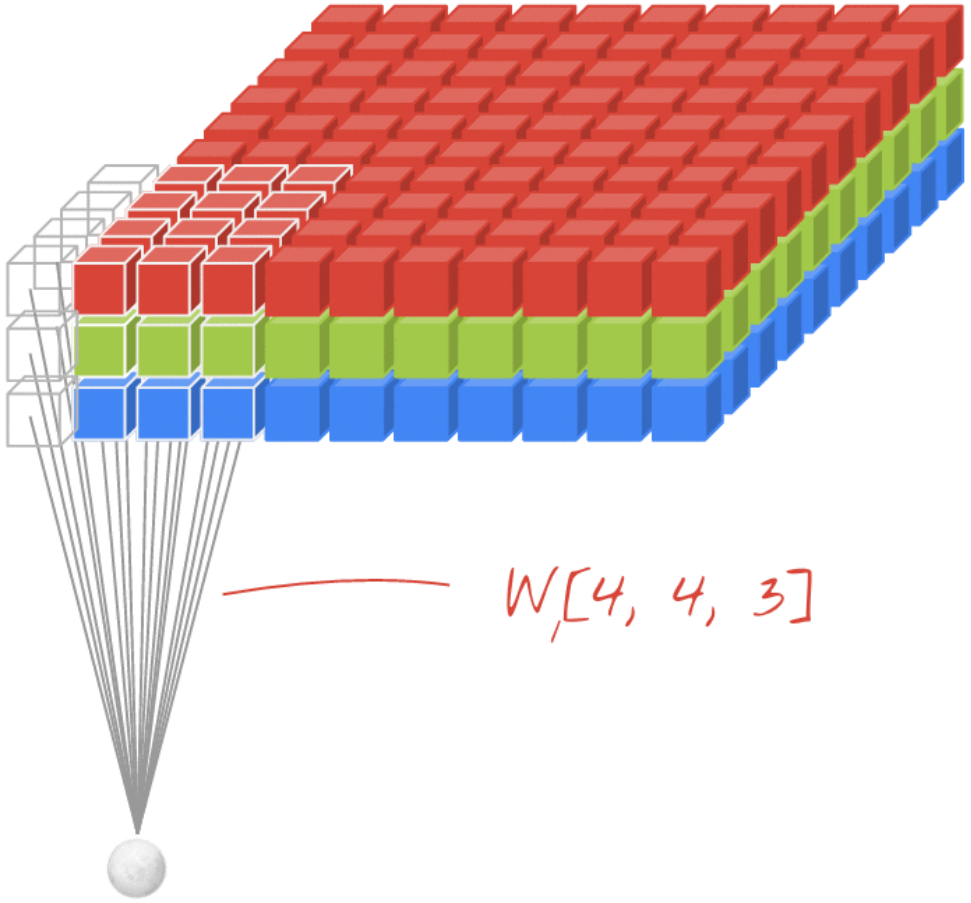
Source: Dr. Francois Fleuret at EPFL

Multi-channel and Multi-kernel 2D Convolution



Source: Dr. Francois Fleuret at EPFL

Multi-channel and Multi-kernel 2D Convolution

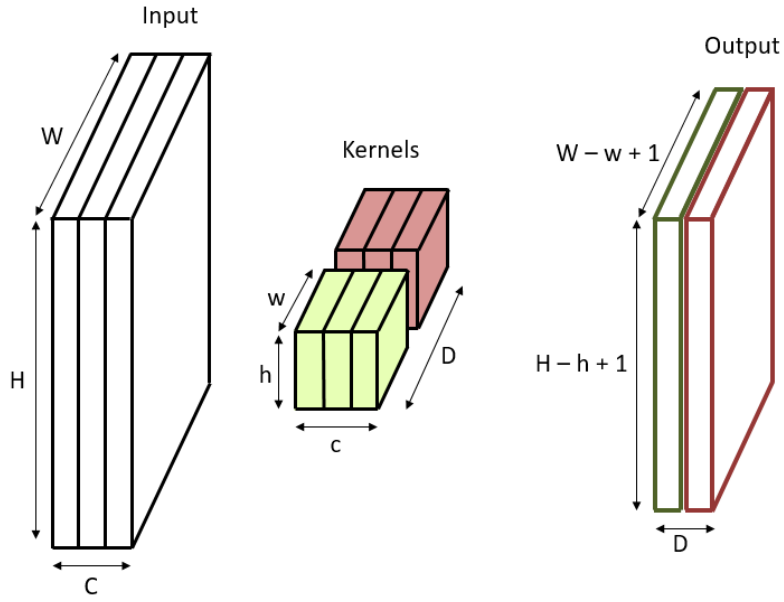


Source: Martin Görner

Multi-channel 2D Convolution

- The kernel is not swiped across channels, just across rows and columns.
- Note that a convolution preserves the signal support structure.
  - A 1D signal is converted into a 1D signal, a 2D signal into a 2D, and neighboring parts of the input signal influence neighboring parts of the output signal.

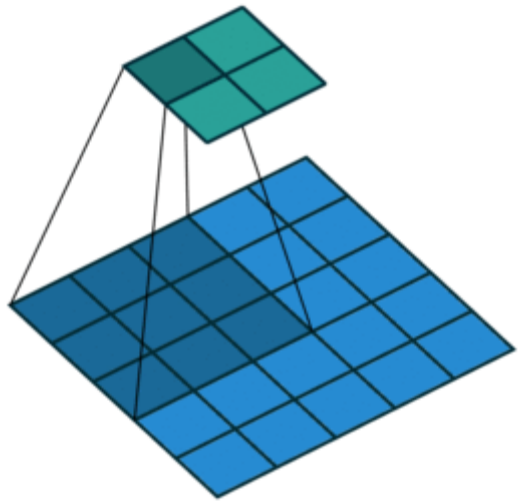
- We usually refer to one of the channels generated by a convolution layer as an activation map.
- The sub-area of an input map that influences a component of the output as the receptive field of the latter.



Padding and Stride

Strides

- Strides: increment step size for the convolution operator
- Reduces the size of the output map

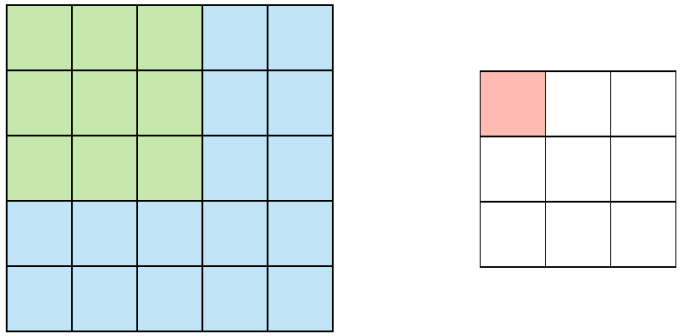


Example with kernel size 3×3 and a stride of 2 (image in blue)

Source: https://github.com/vdumoulin/conv_arithmetic

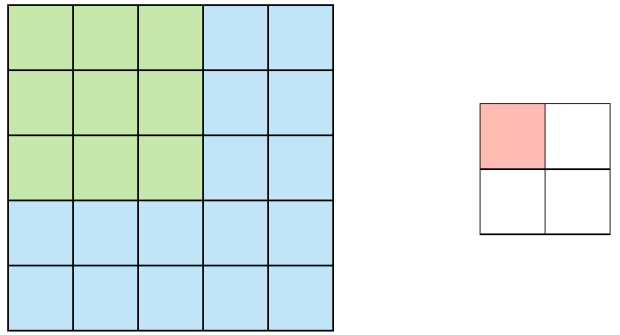
Strides

- Strides: increment step size for the convolution operator
- Reduces the size of the output map



Stride 1 Feature Map

Example with kernel size 3×3 and a stride of 1



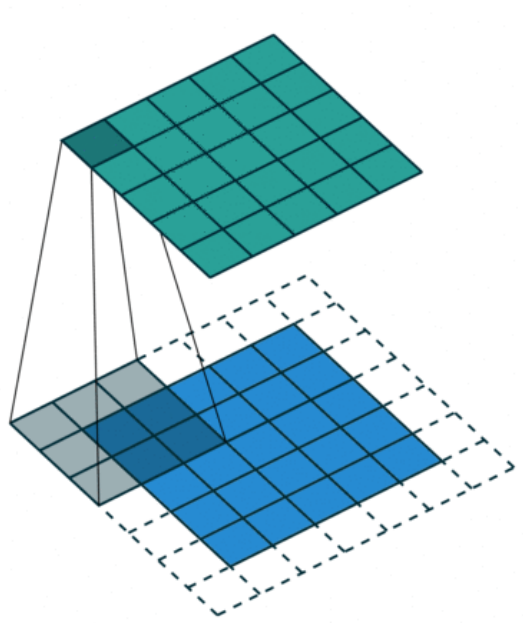
Stride 2 Feature Map

Example with kernel size 3×3 and a stride of 2

Source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks

Padding

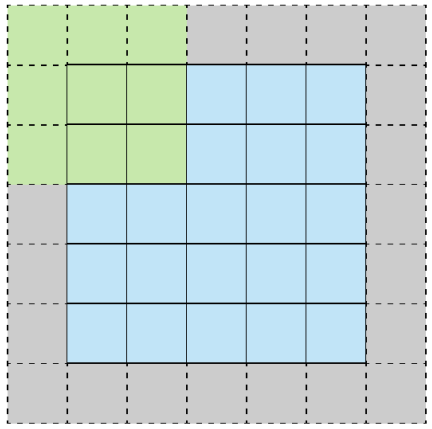
- Padding: artificially fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields
- Usually fill with 0s



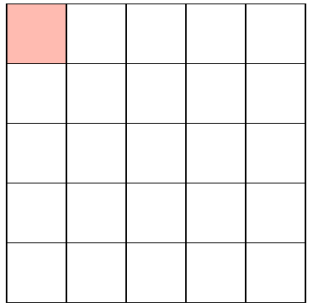
Source: https://github.com/vdumoulin/conv_arithmetic

Padding

- Padding: artificially fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields
- Usually fill with 0s



Stride 1 with Padding

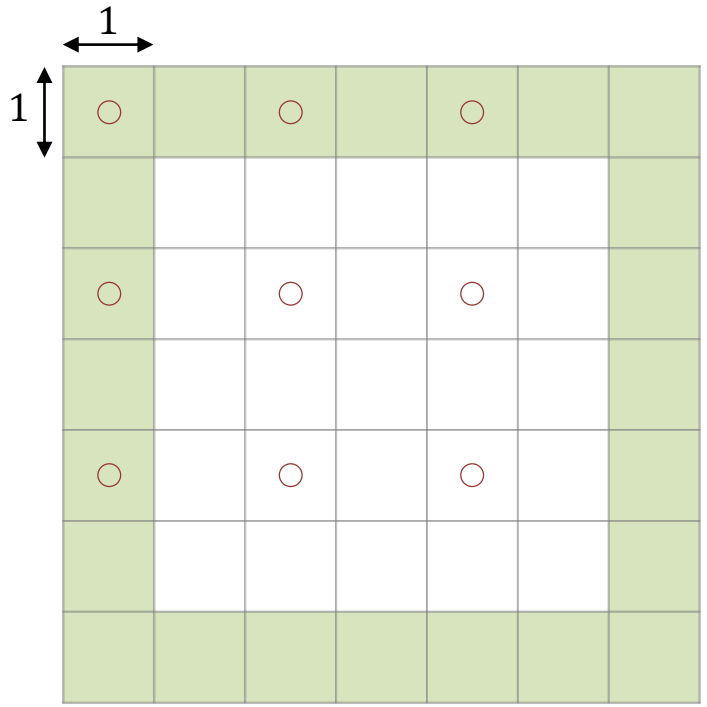


Feature Map

Source: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks

Padding and Stride

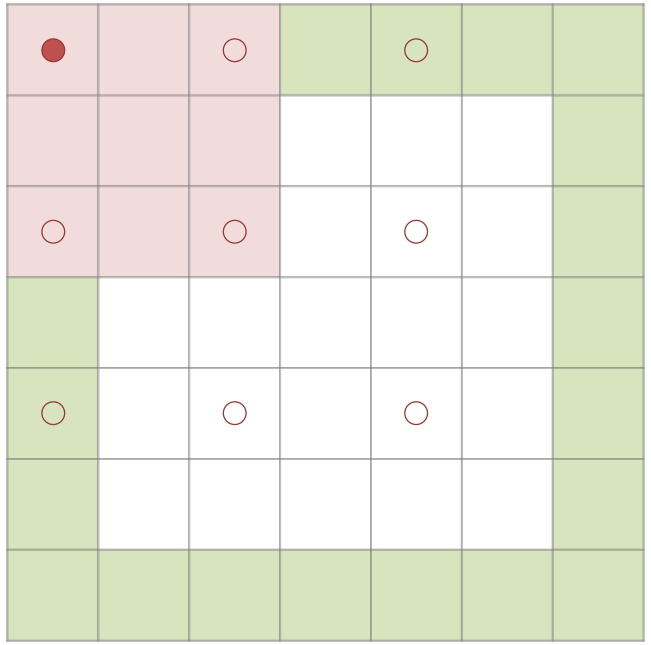
- Here with $5 \times 5 \times C$ as input, a padding of $(1\ ,1)$, a stride of $(2\ ,2)$



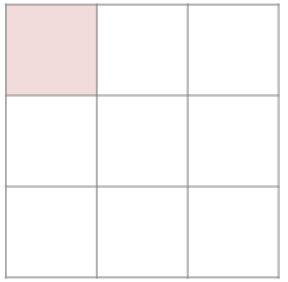
Input

Padding and Stride

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



Input

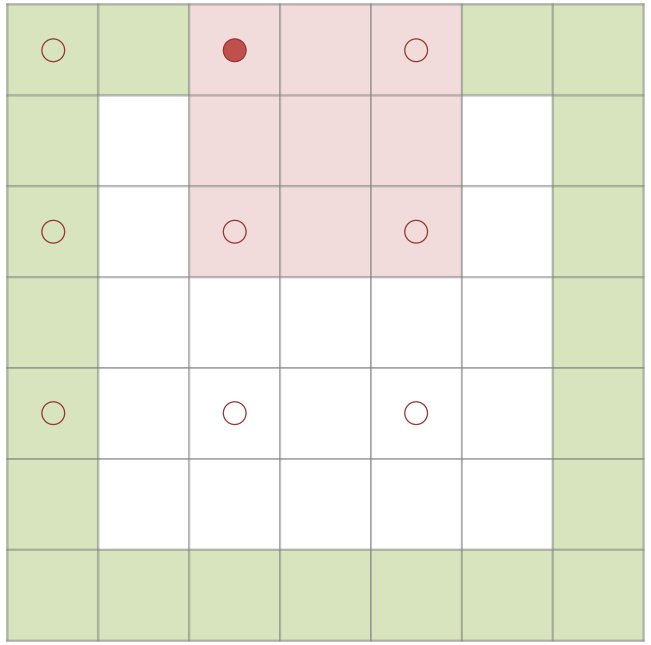


Output

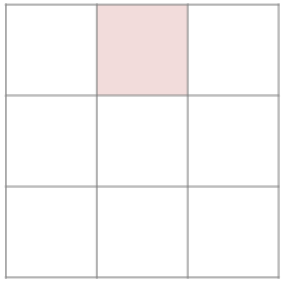
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

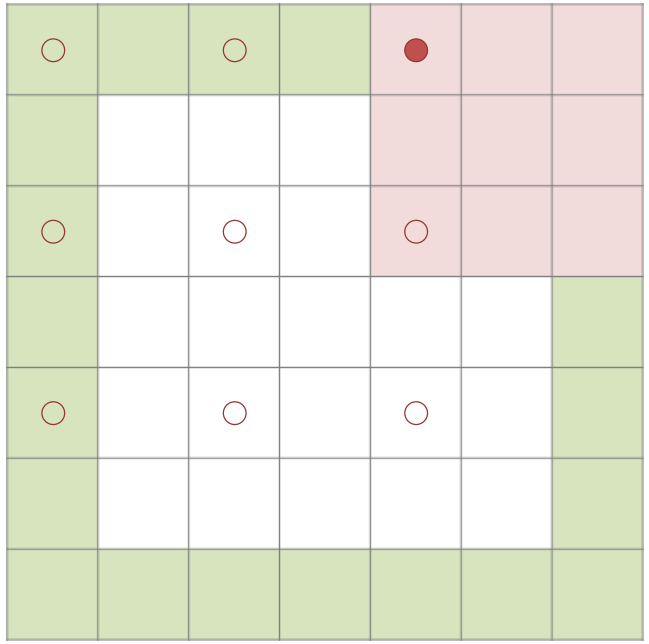


Output

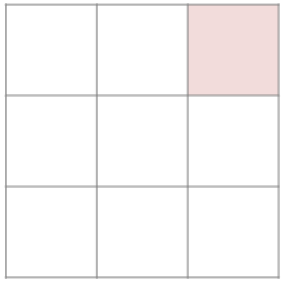
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

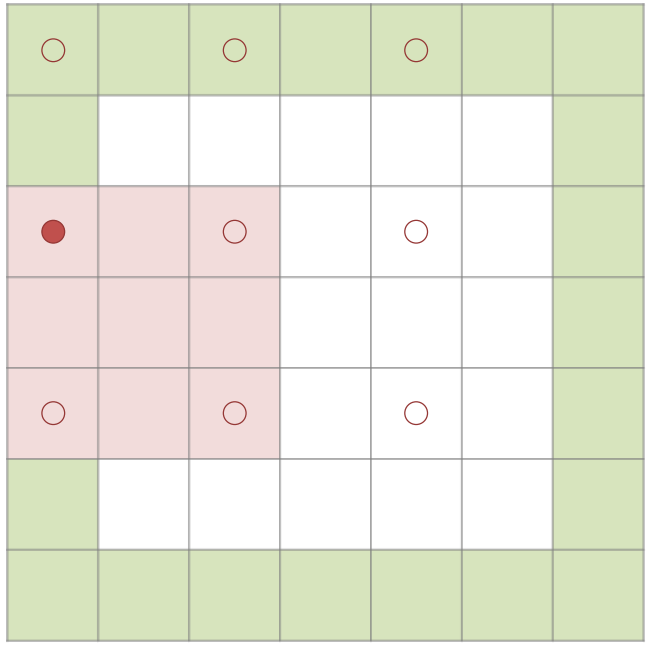


Output

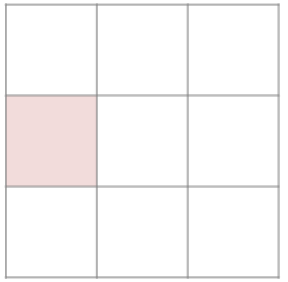
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

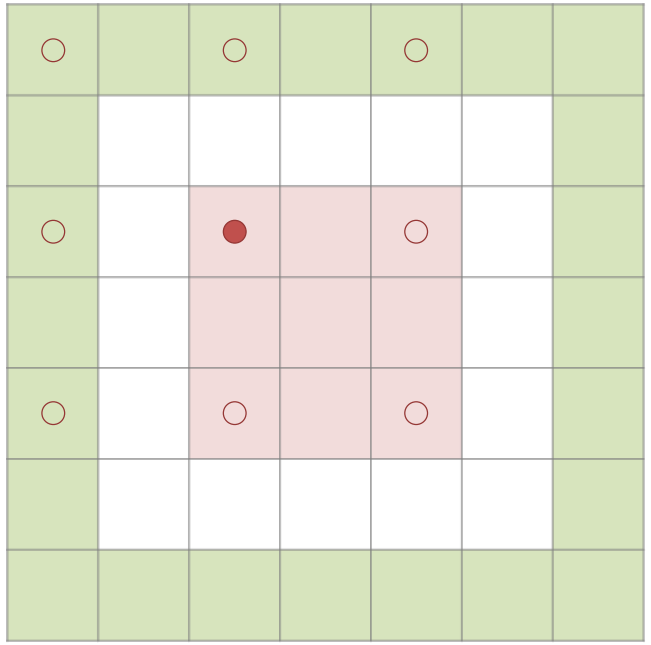


Output

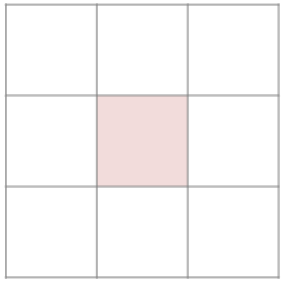
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

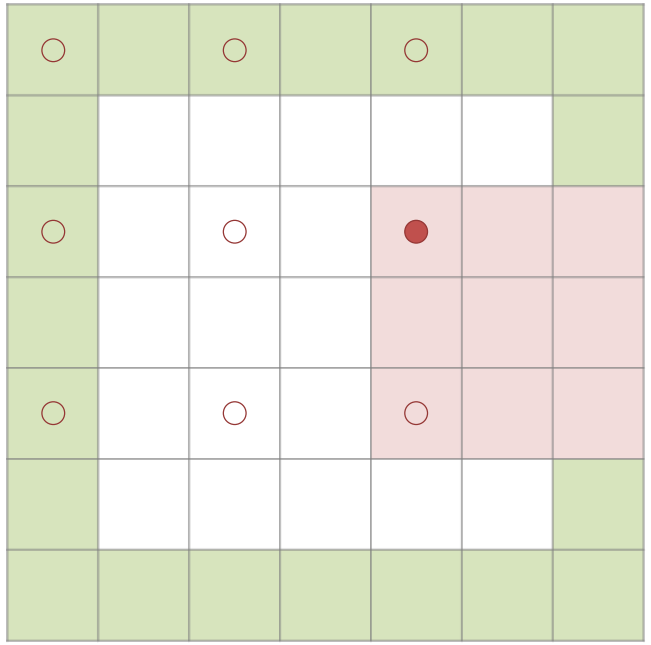


Output

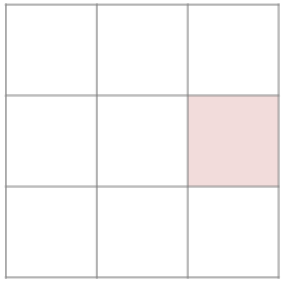
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

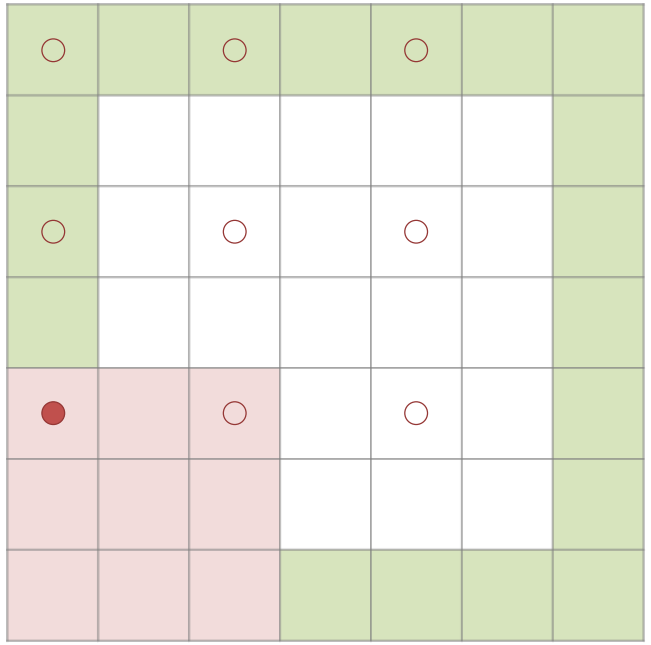


Output

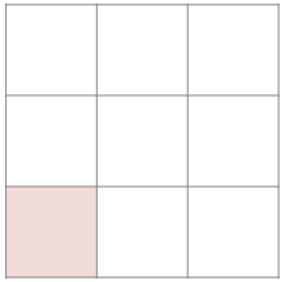
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

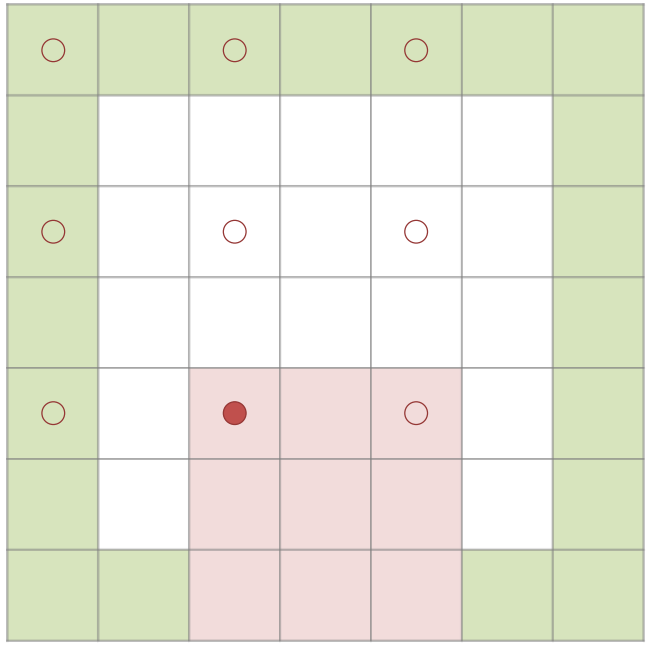


Output

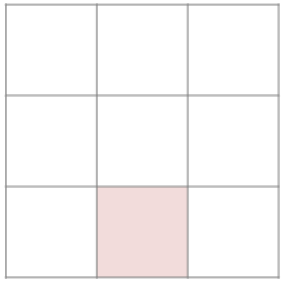
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



Input

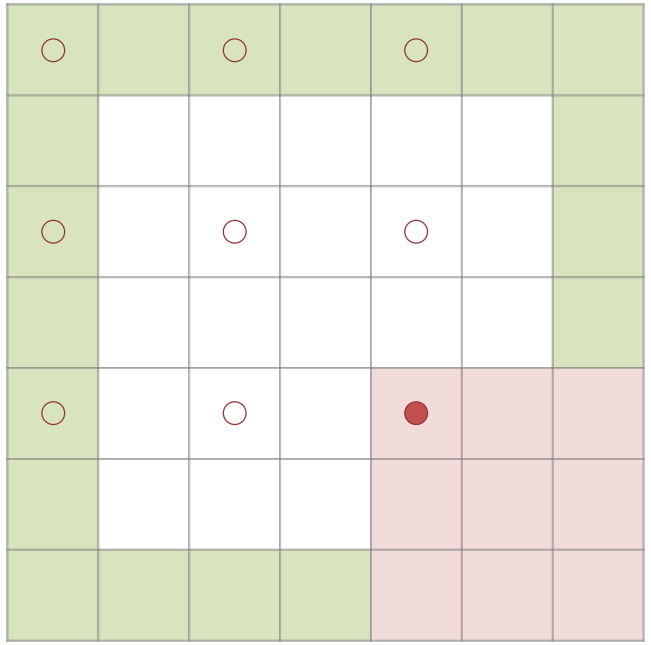


Output

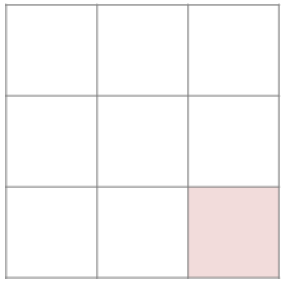
Source: Dr. Francois Fleuret at EPFL

Padding and Stride

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



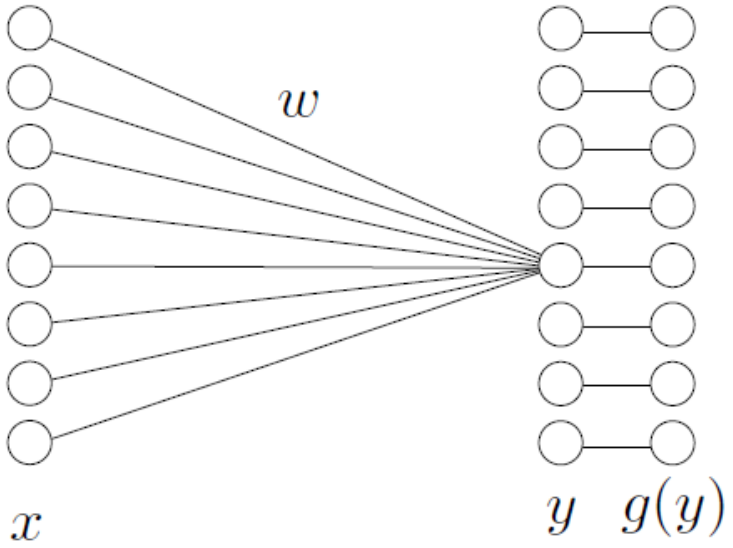
Input

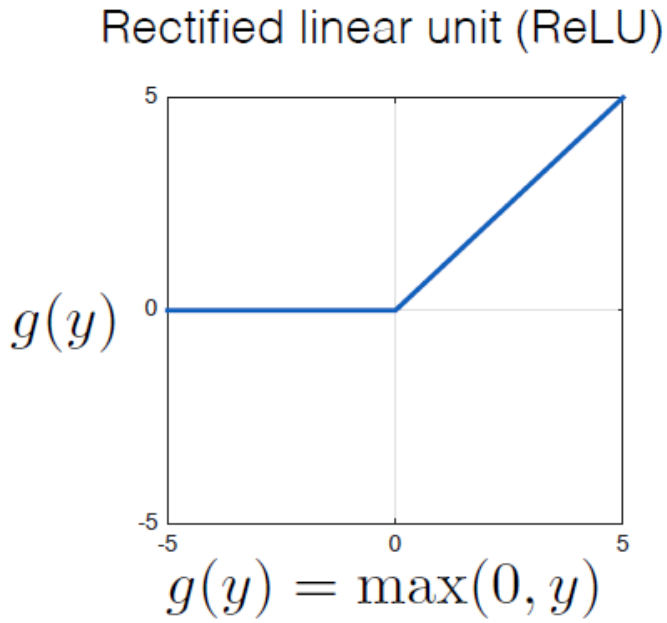


Output

Source: Dr. Francois Fleuret at EPFL

Nonlinear Activation Function

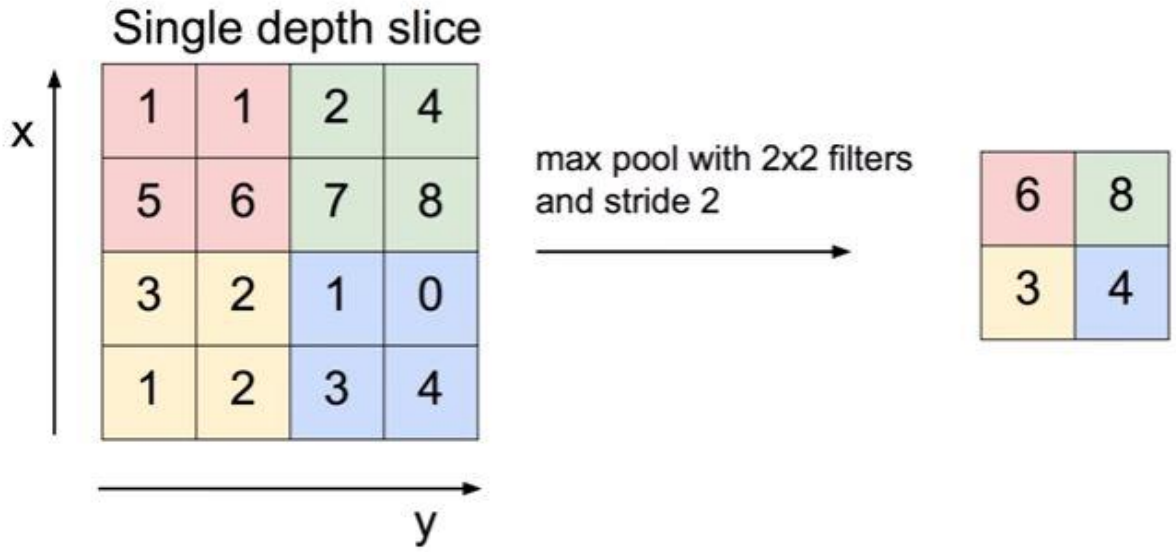




Pooling

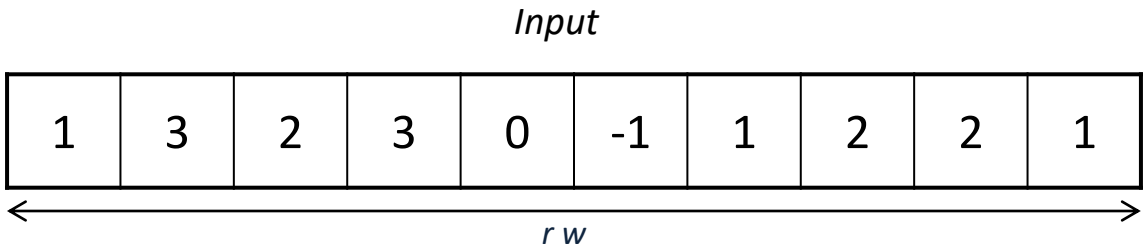
Pooling

- Compute a maximum value in a sliding window (max pooling)
- Reduce spatial resolution for faster computation
- Achieve invariance to local translation
- Max pooling introduces invariances
  - Pooling size : 2×2
  - No parameters: max or average of 2x2 units

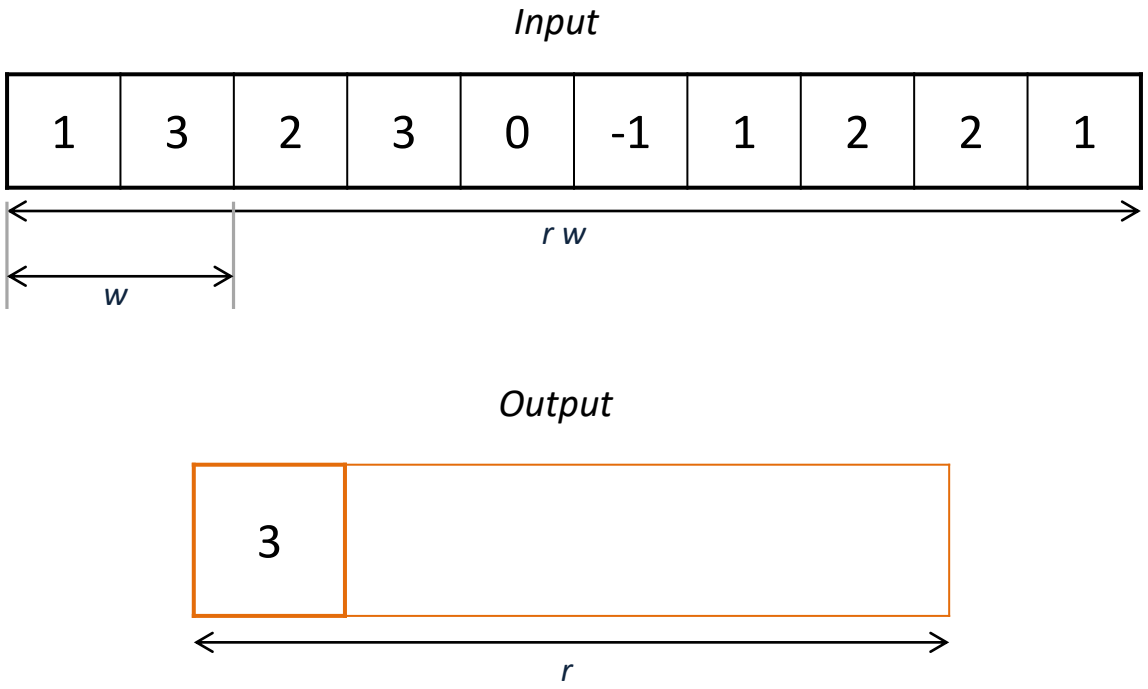


Pooling

- The most standard type of pooling is the max-pooling, which computes max values over non-overlapping blocks
- For instance in 1D with a window of size 2

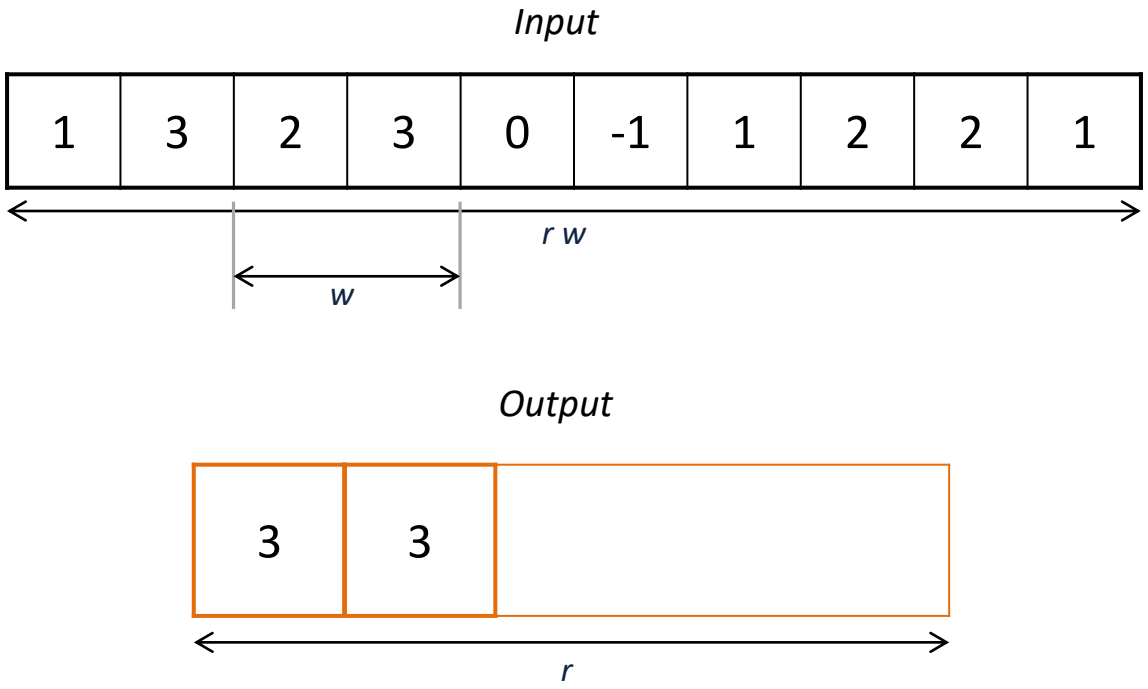


Pooling



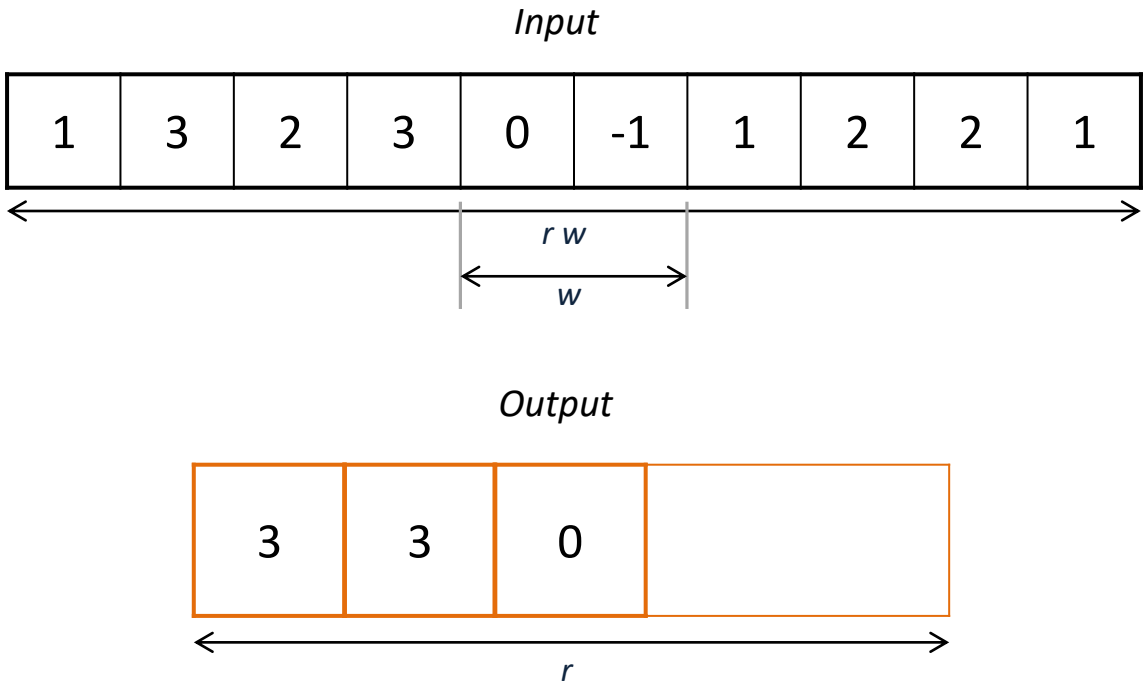
Source: Dr. Francois Fleuret at EPFL

Pooling



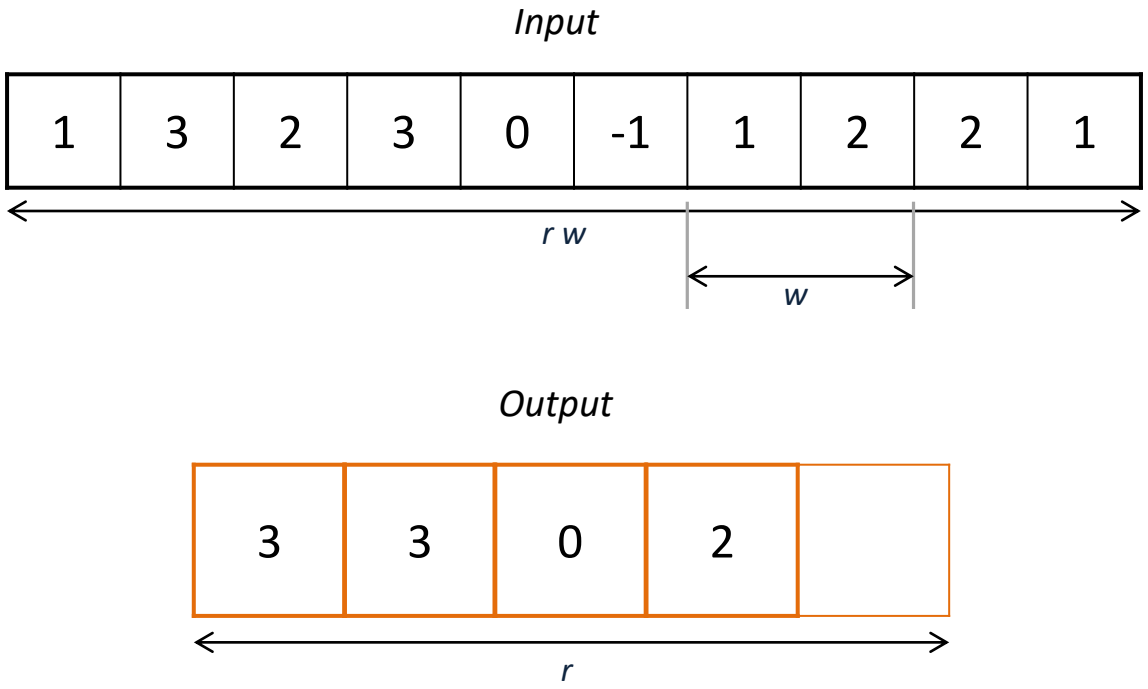
Source: Dr. Francois Fleuret at EPFL

Pooling



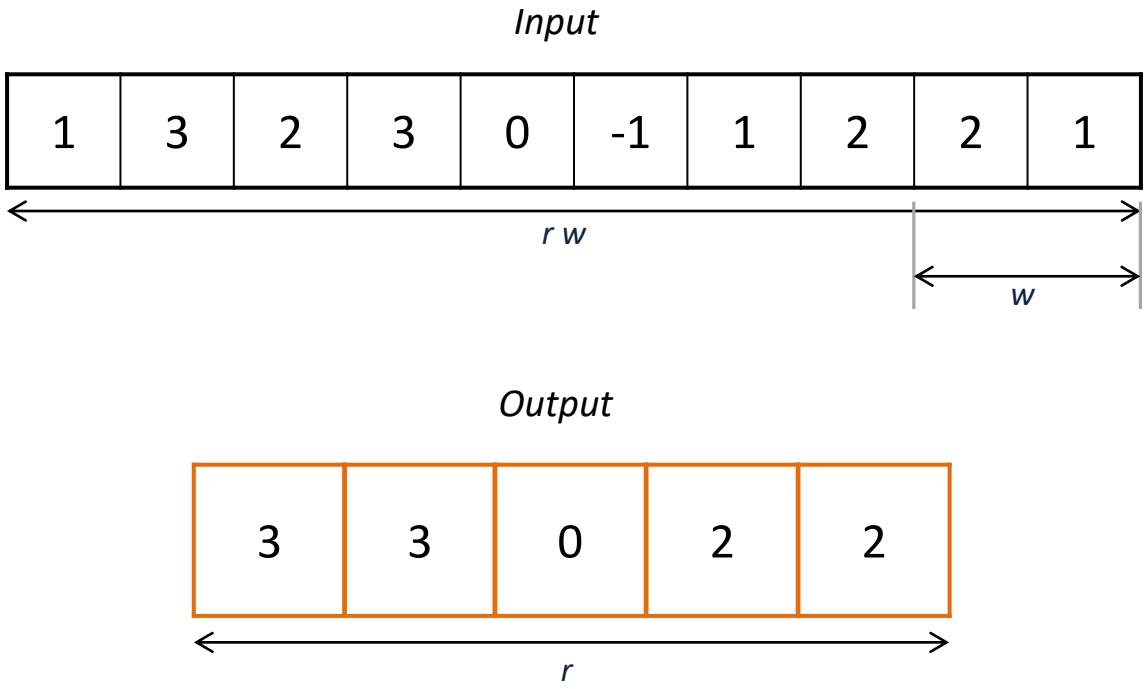
Source: Dr. Francois Fleuret at EPFL

Pooling



Source: Dr. Francois Fleuret at EPFL

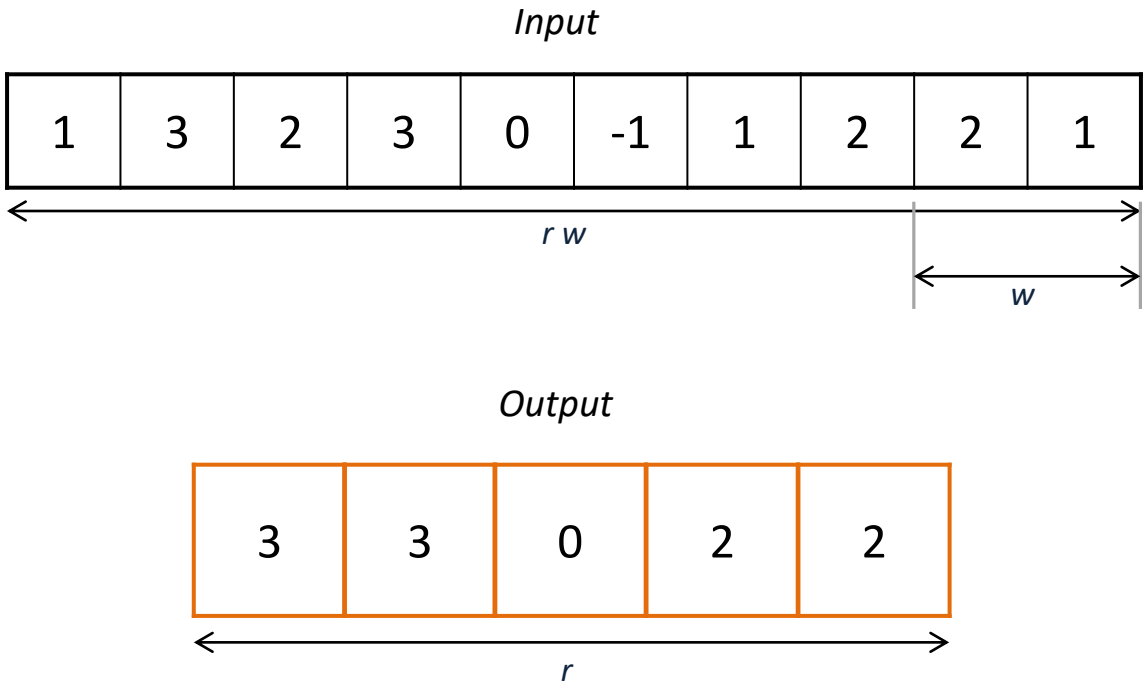
Pooling



Source: Dr. Francois Fleuret at EPFL

Pooling

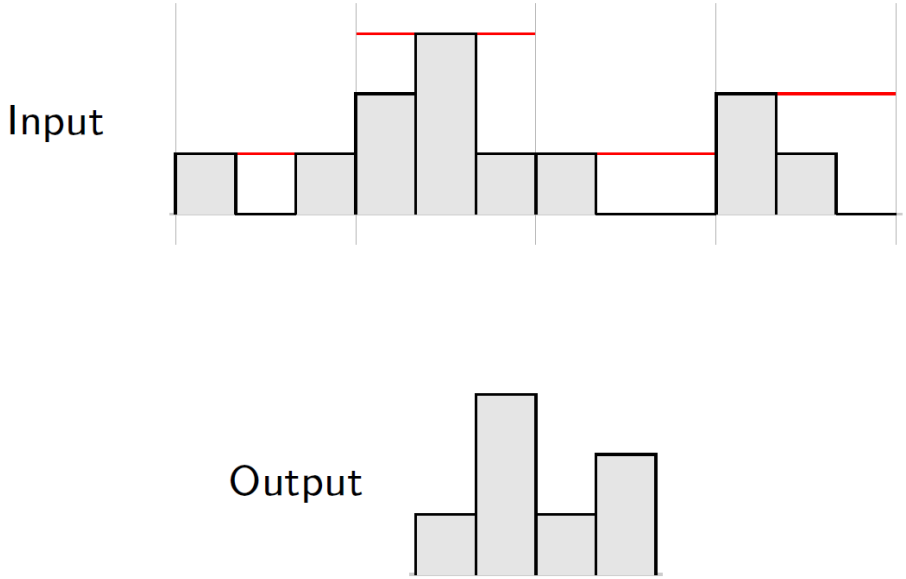
- Such an operation aims at grouping several activations into a single “more meaningful” one.



- The average pooling computes average values per block instead of max values

Pooling: Invariance

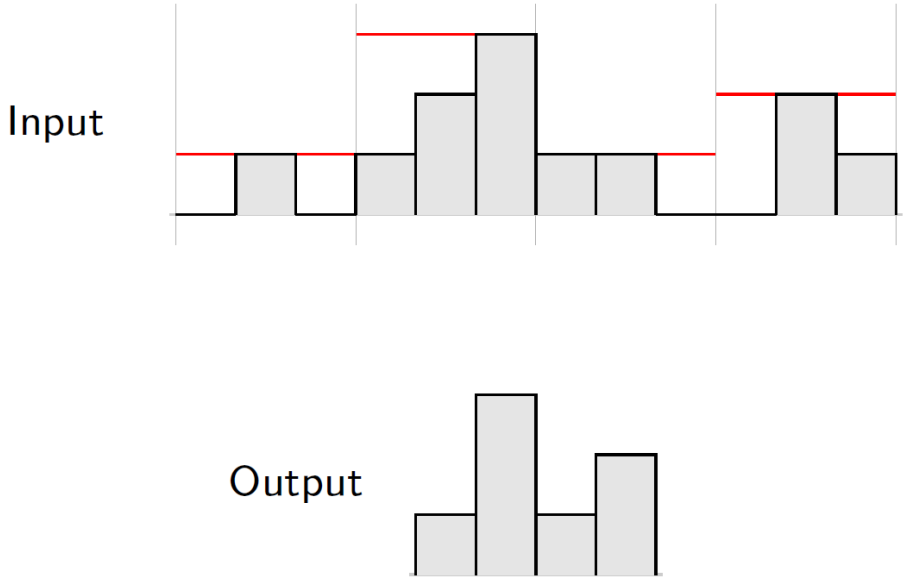
- Pooling provides invariance to any permutation inside one of the cell
- More practically, it provides a pseudo-invariance to deformations that result into local translations



Source: Dr. Francois Fleuret at EPFL

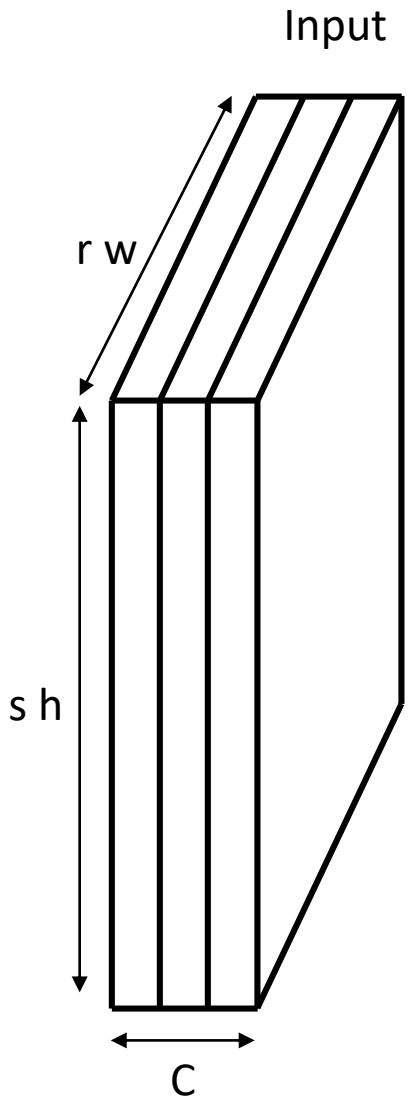
Pooling: Invariance

- Pooling provides invariance to any permutation inside one of the cell
- More practically, it provides a pseudo-invariance to deformations that result into local translations



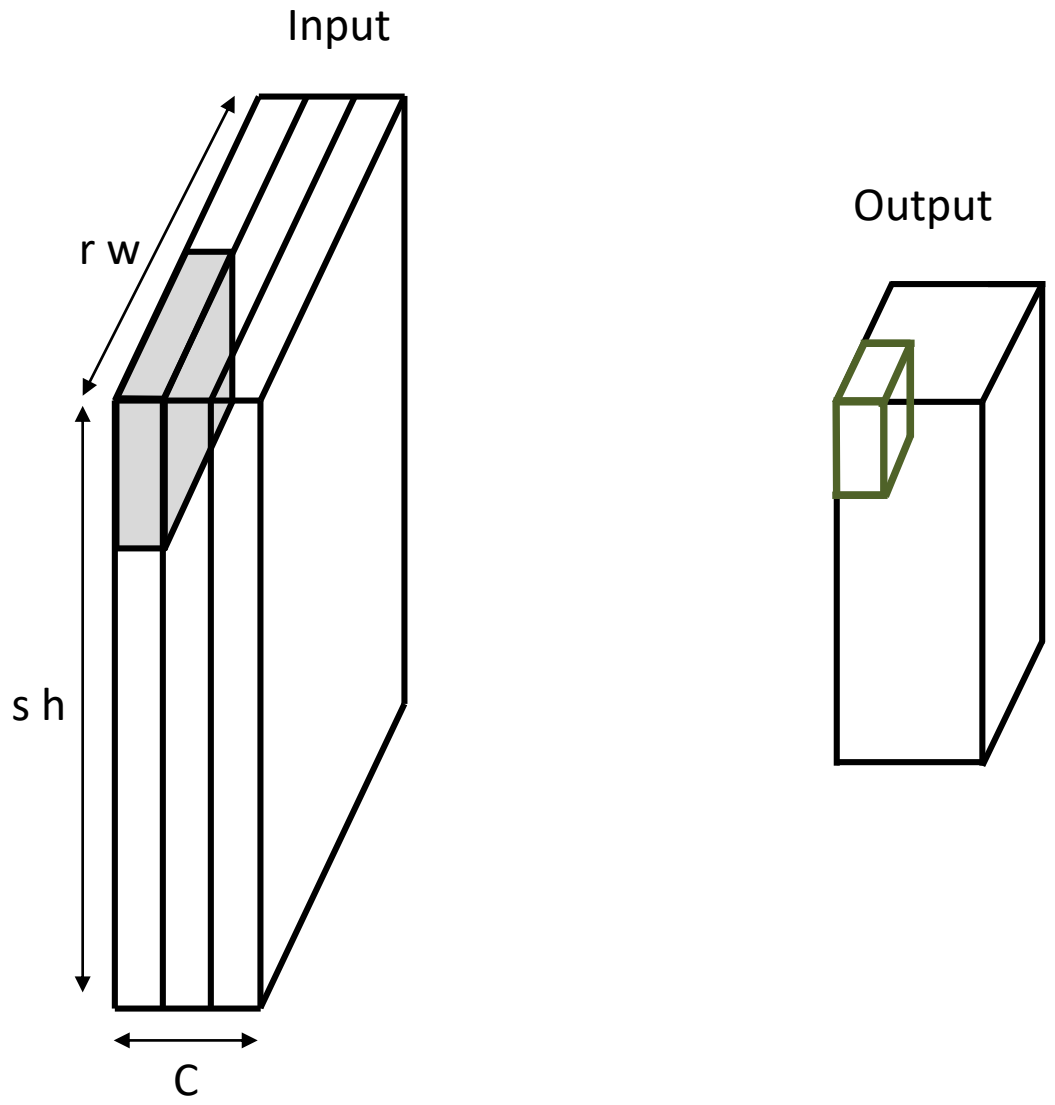
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling

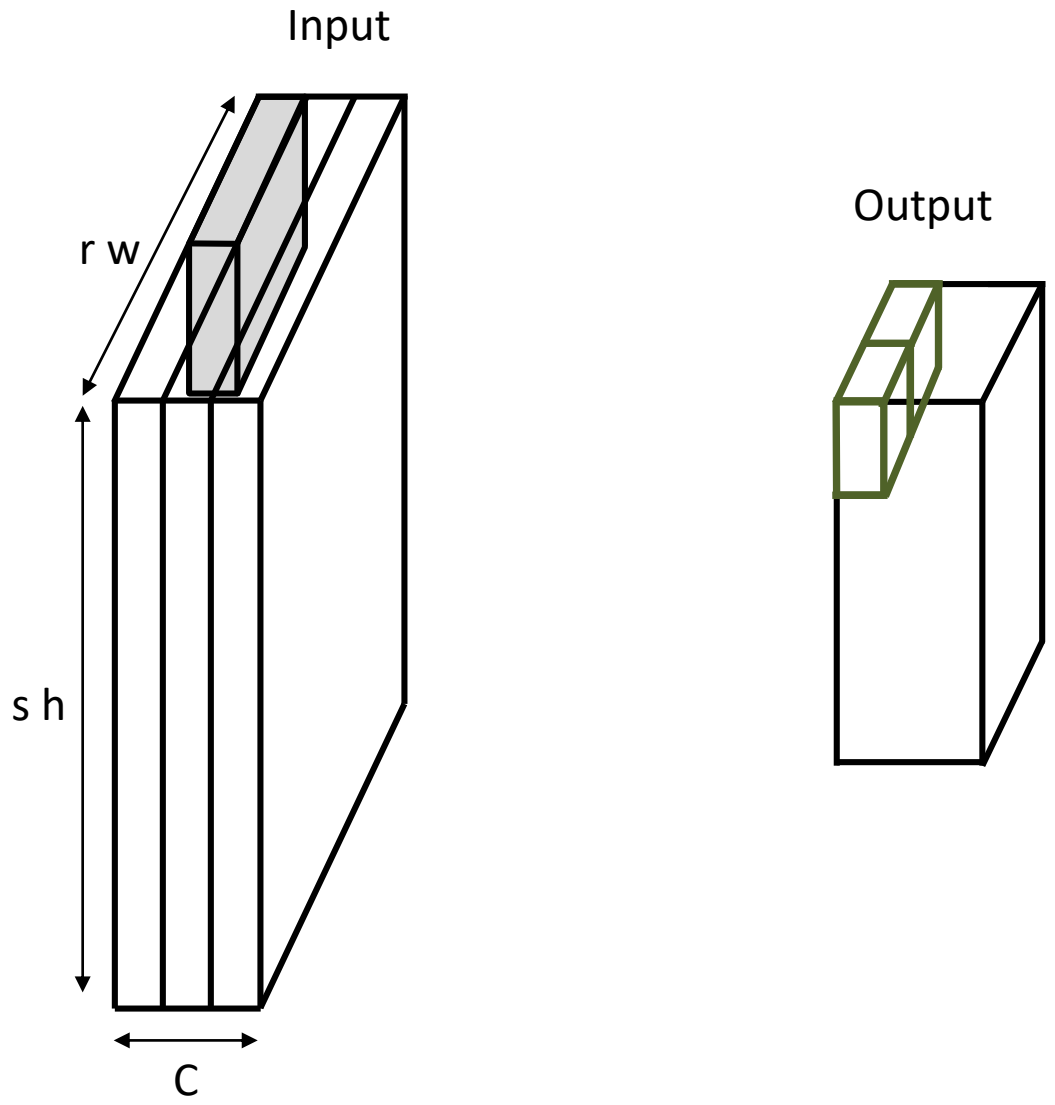


Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling

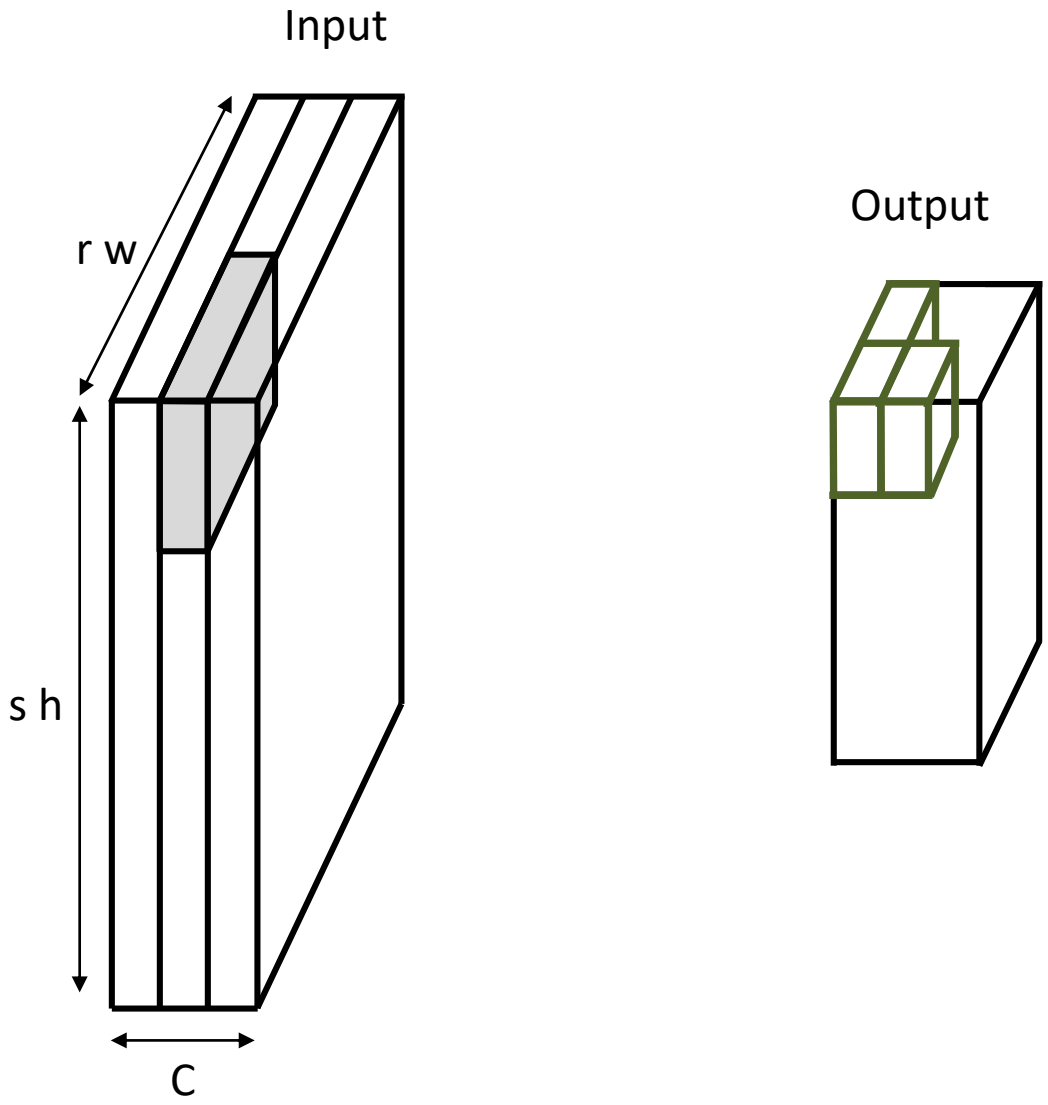


Multi-channel Pooling



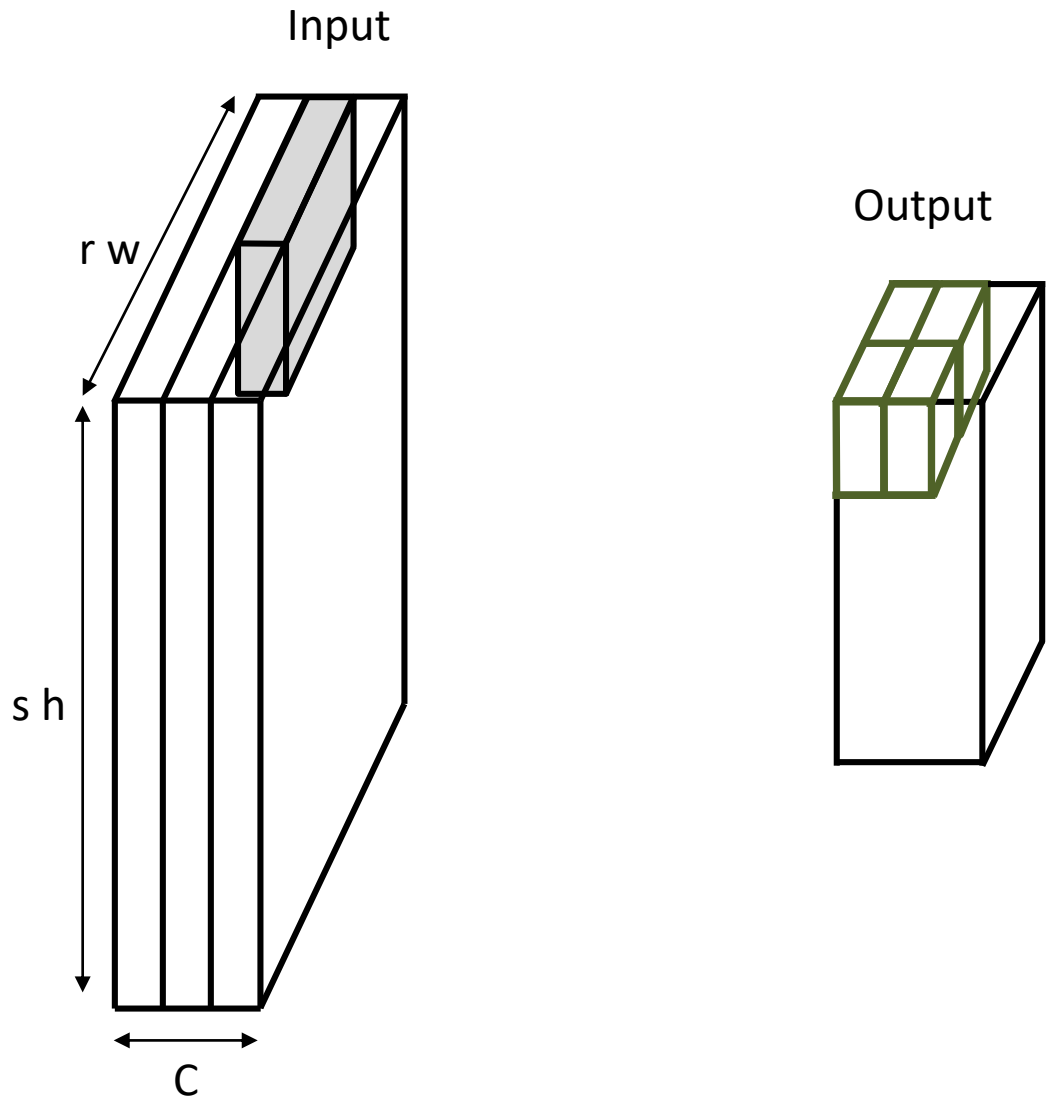
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



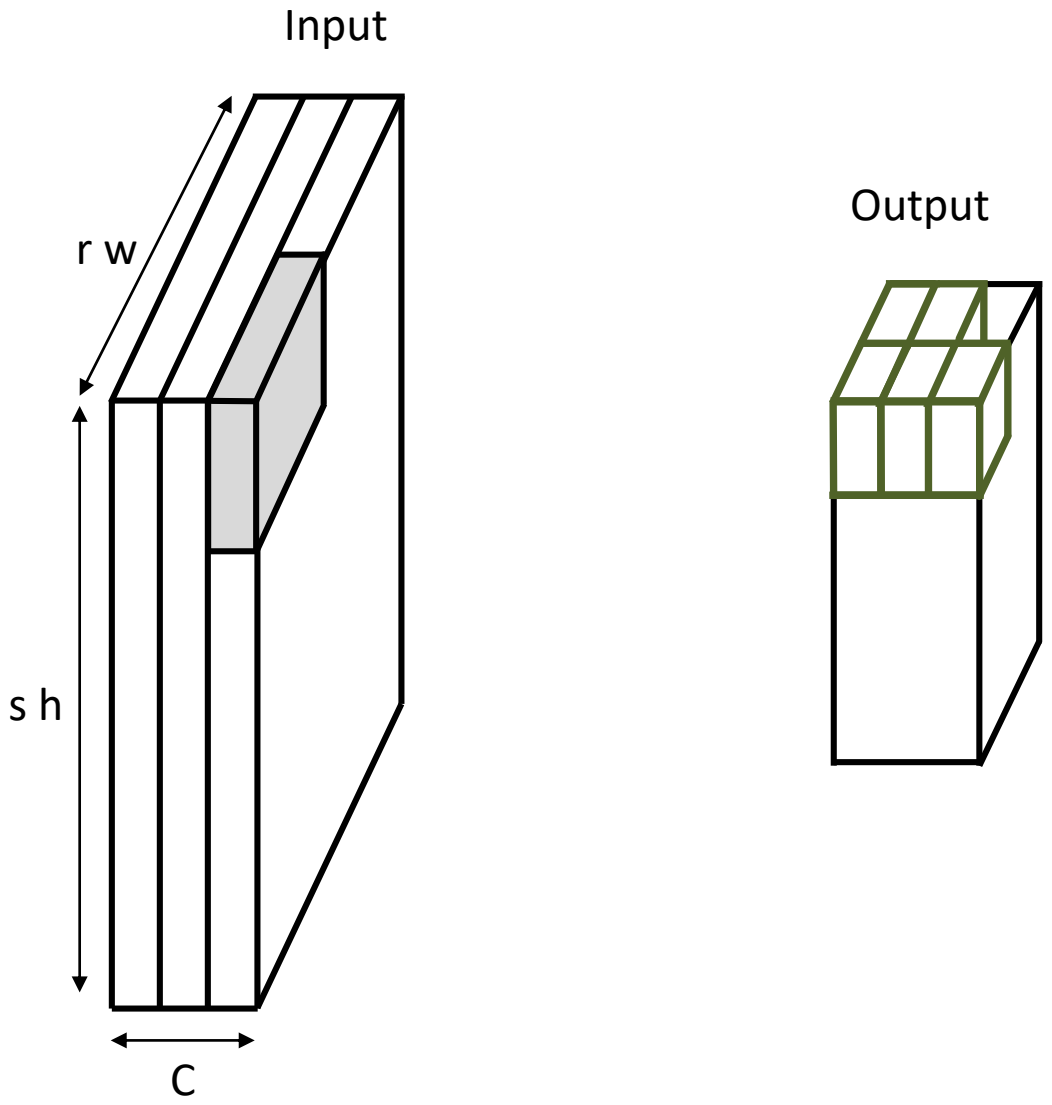
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



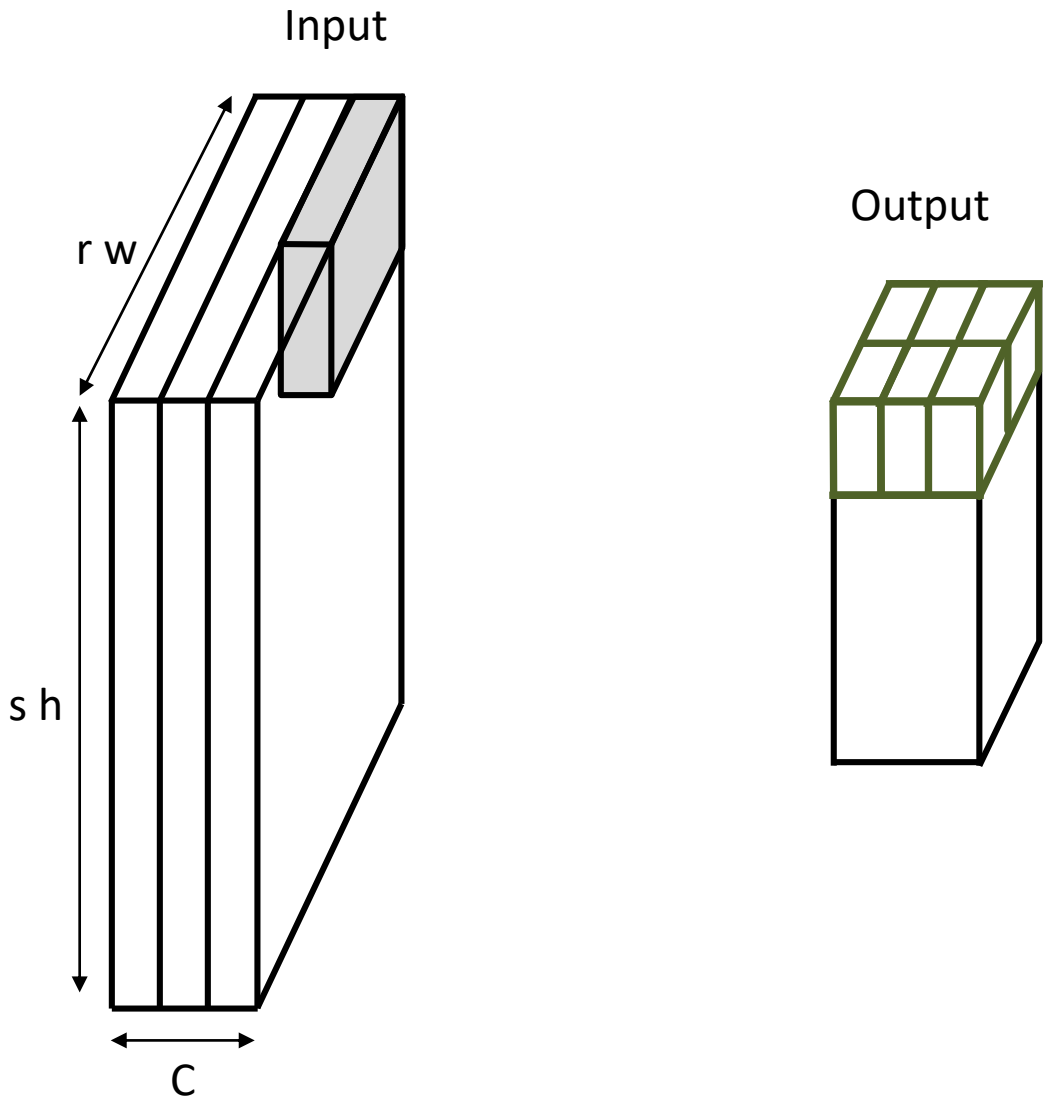
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



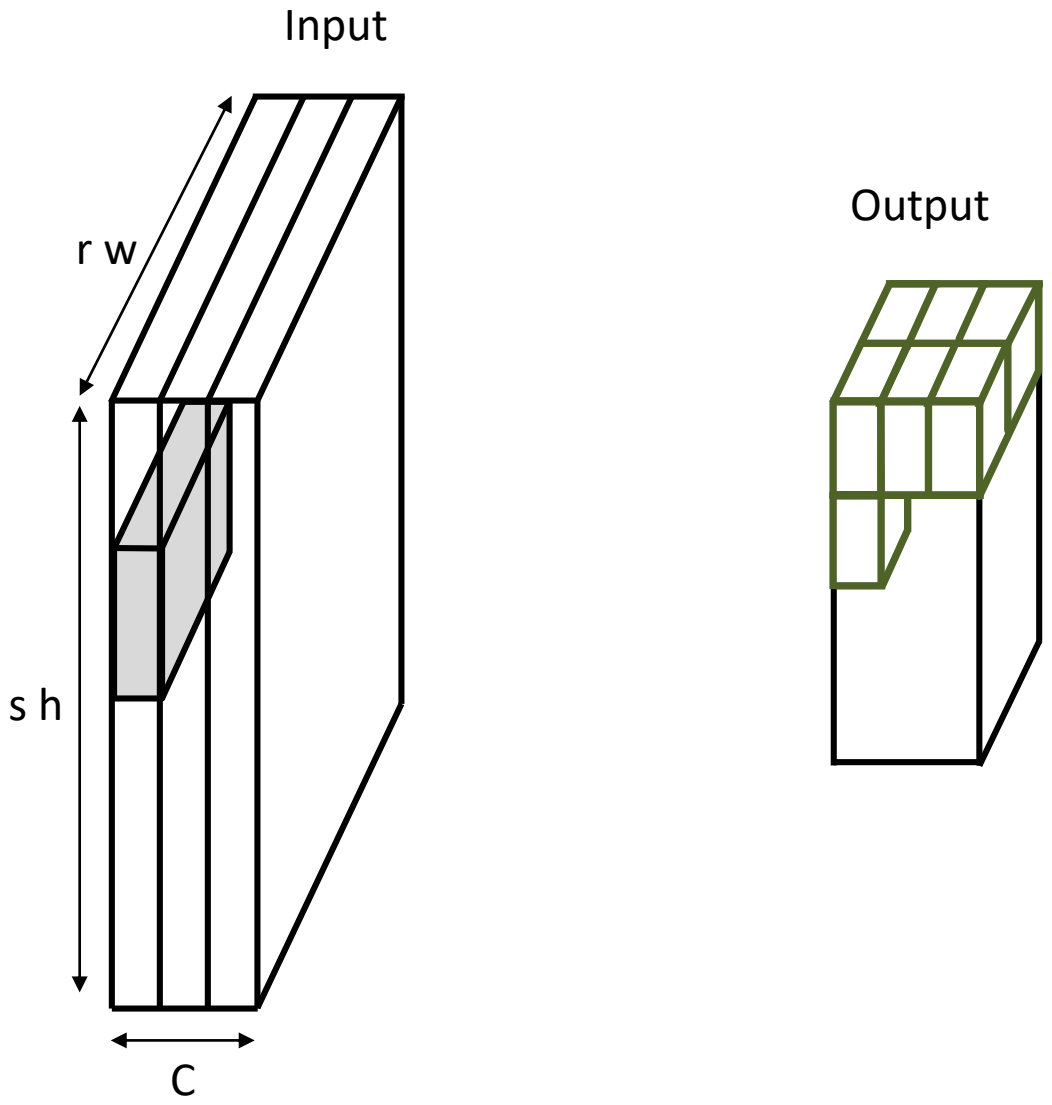
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



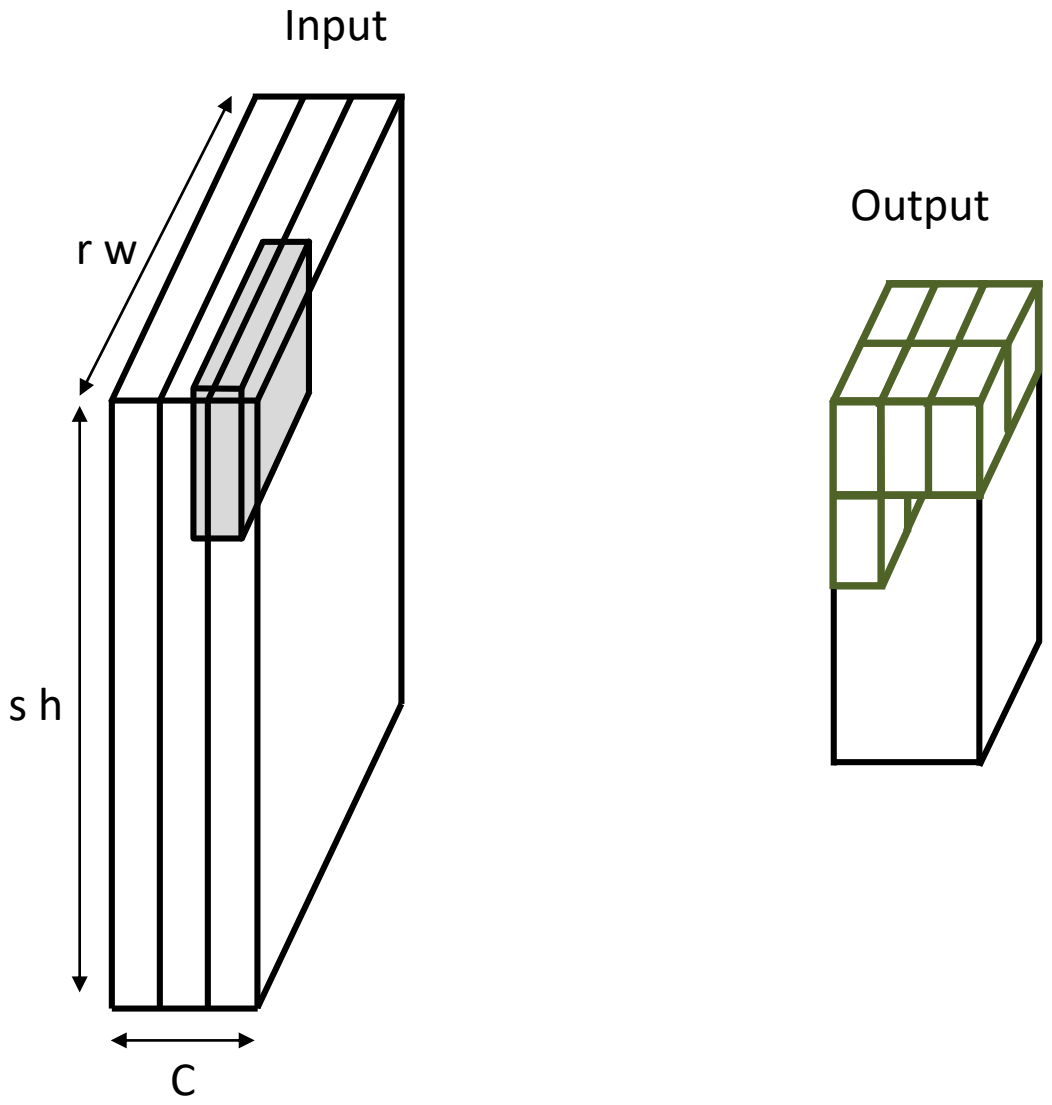
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



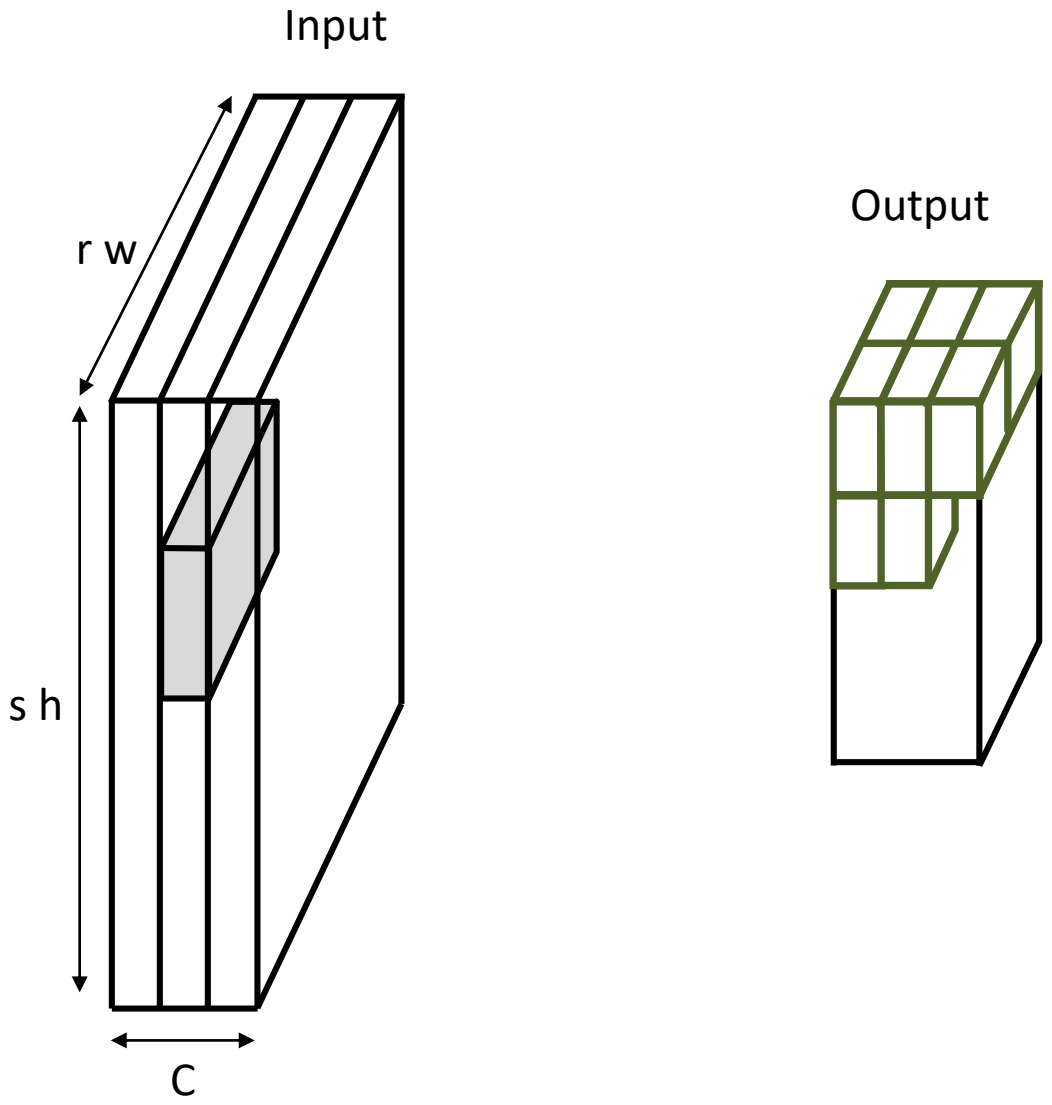
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



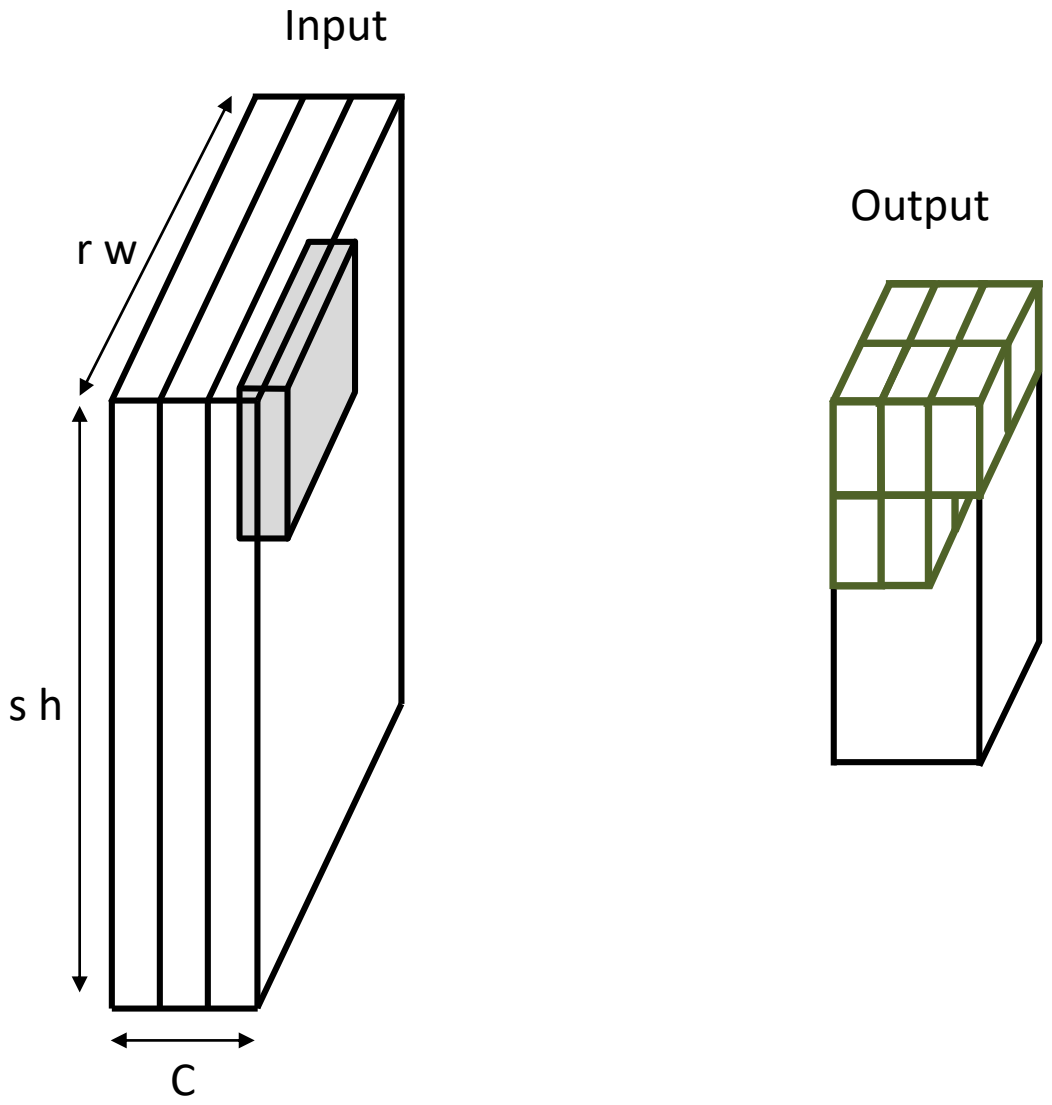
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



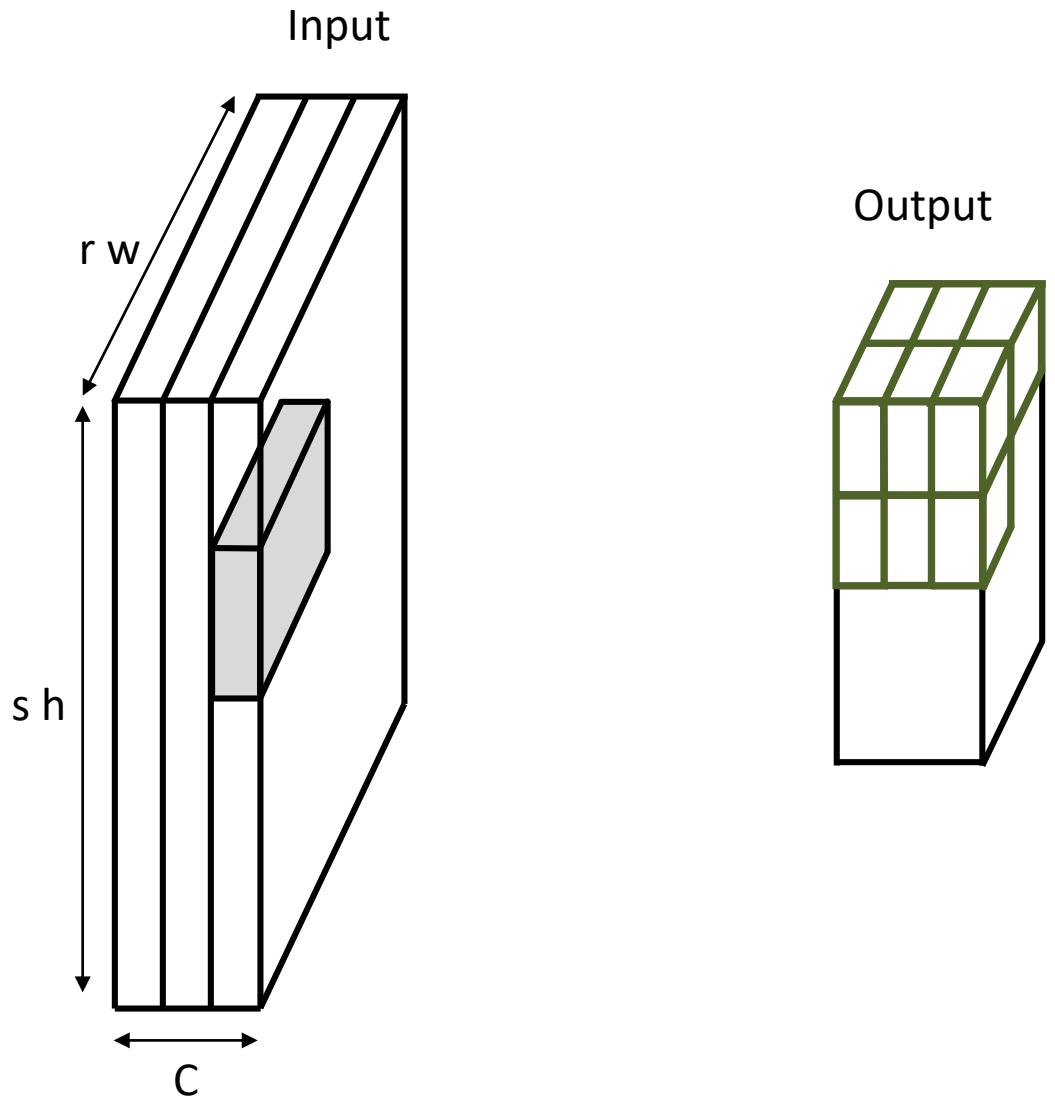
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



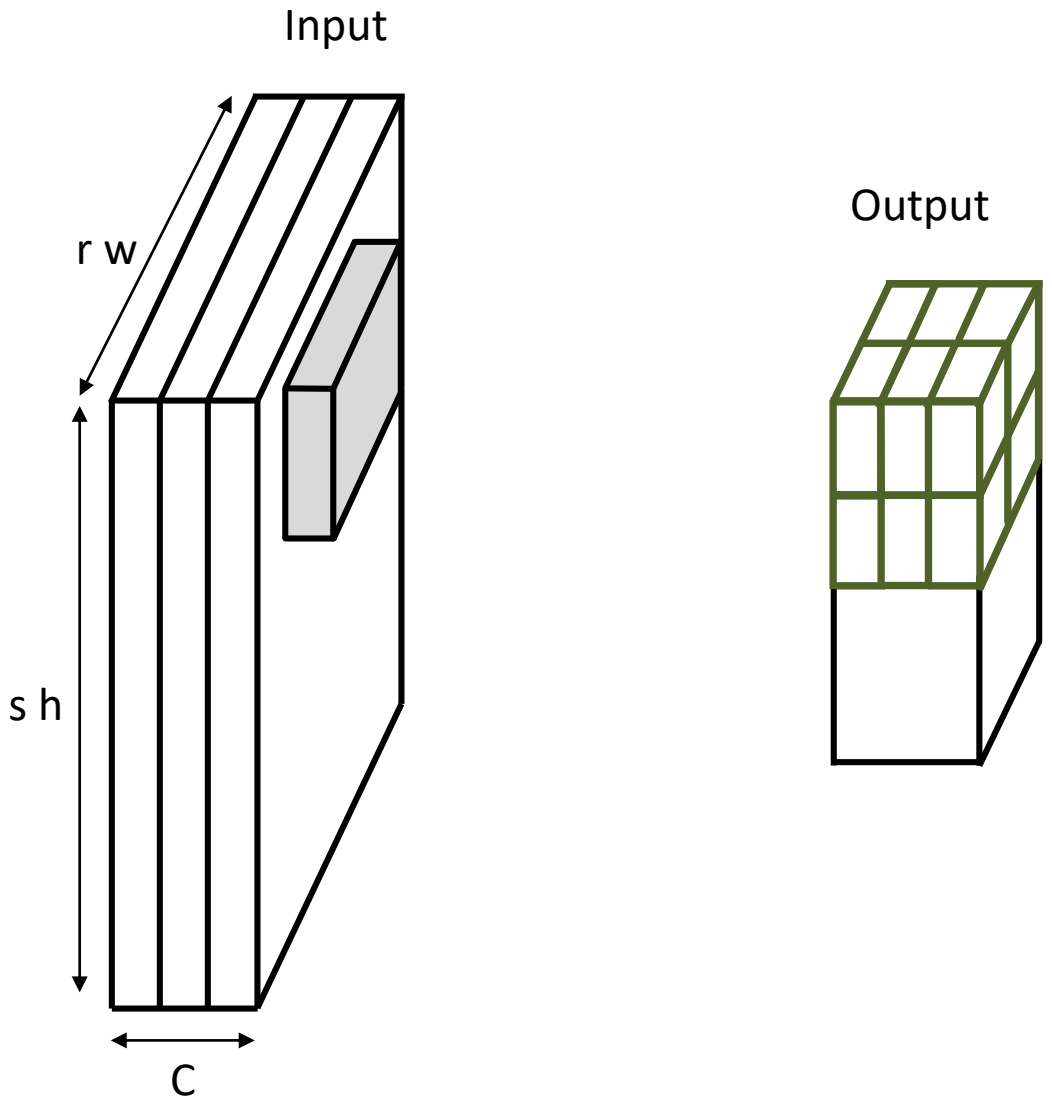
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



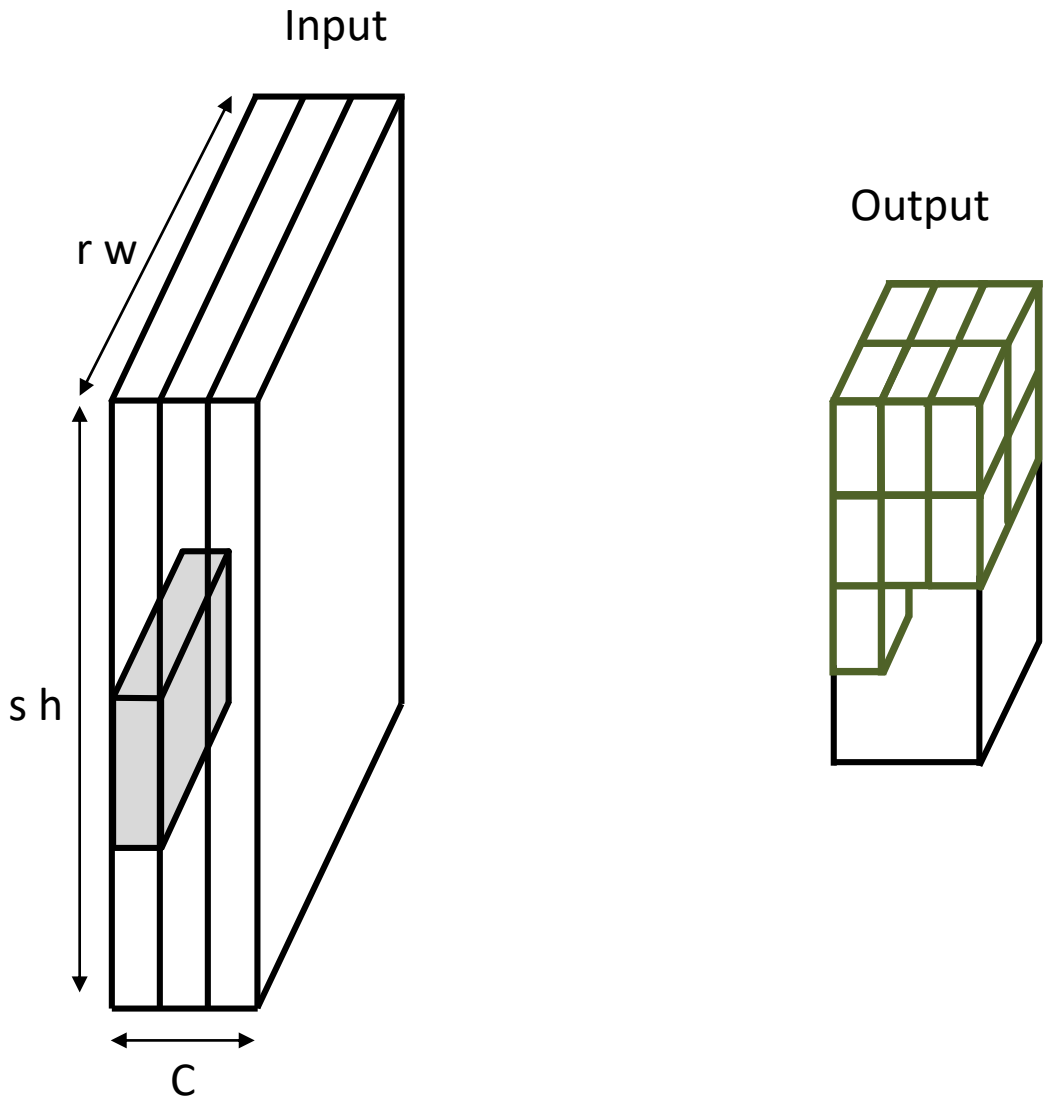
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



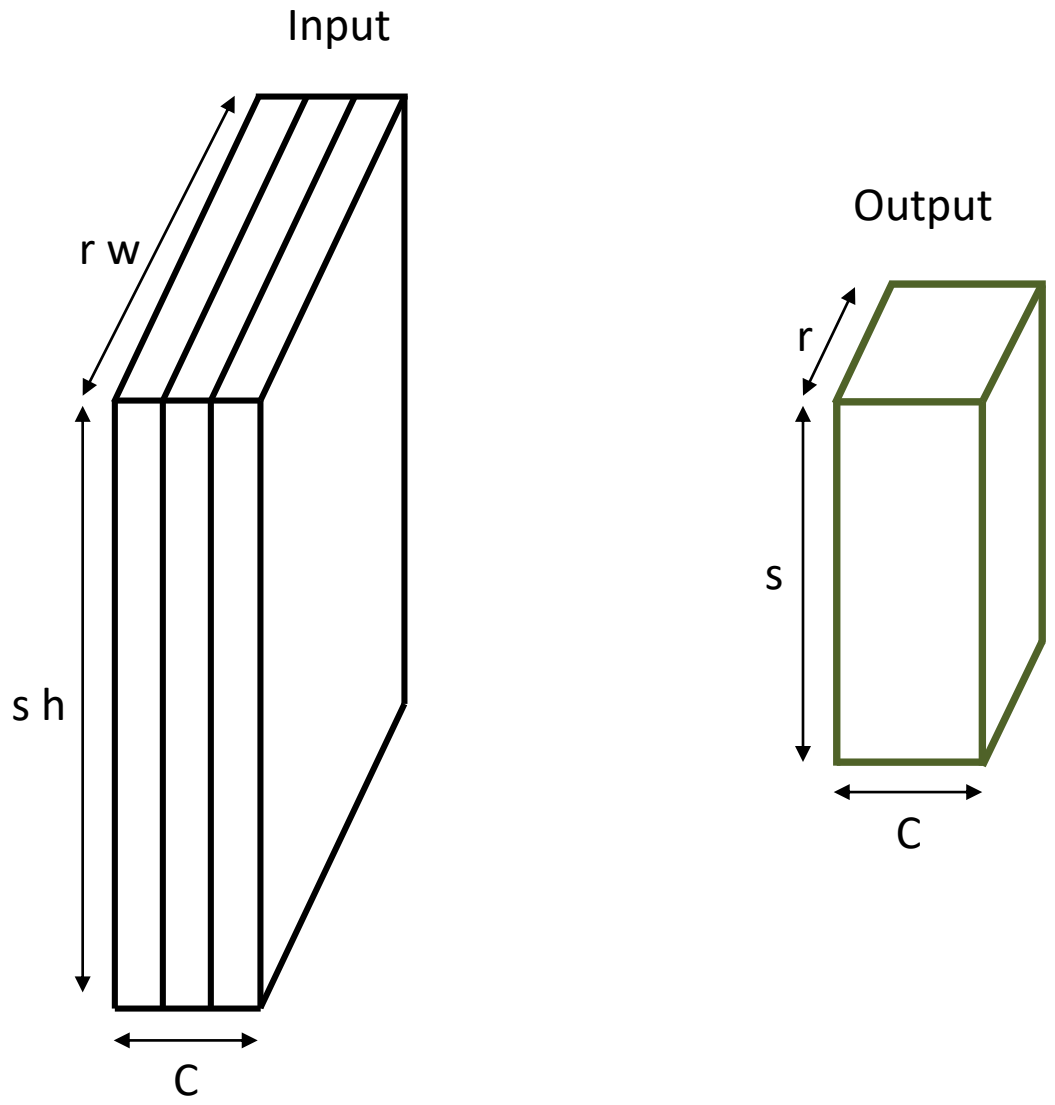
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling



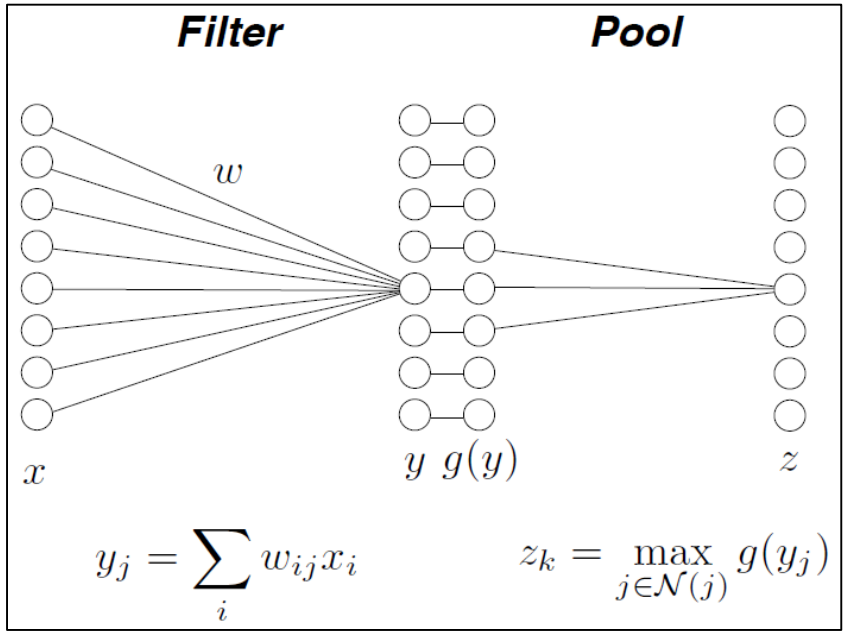
Source: Dr. Francois Fleuret at EPFL

Multi-channel Pooling

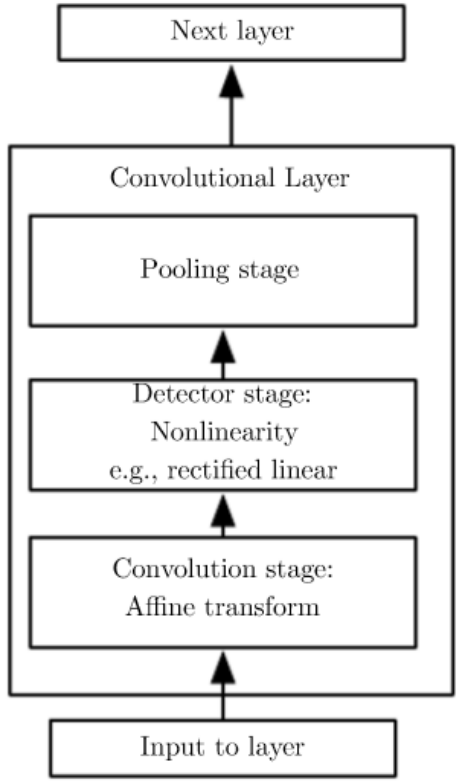


Source: Dr. Francois Fleuret at EPFL

Inside the Convolution Layer Block

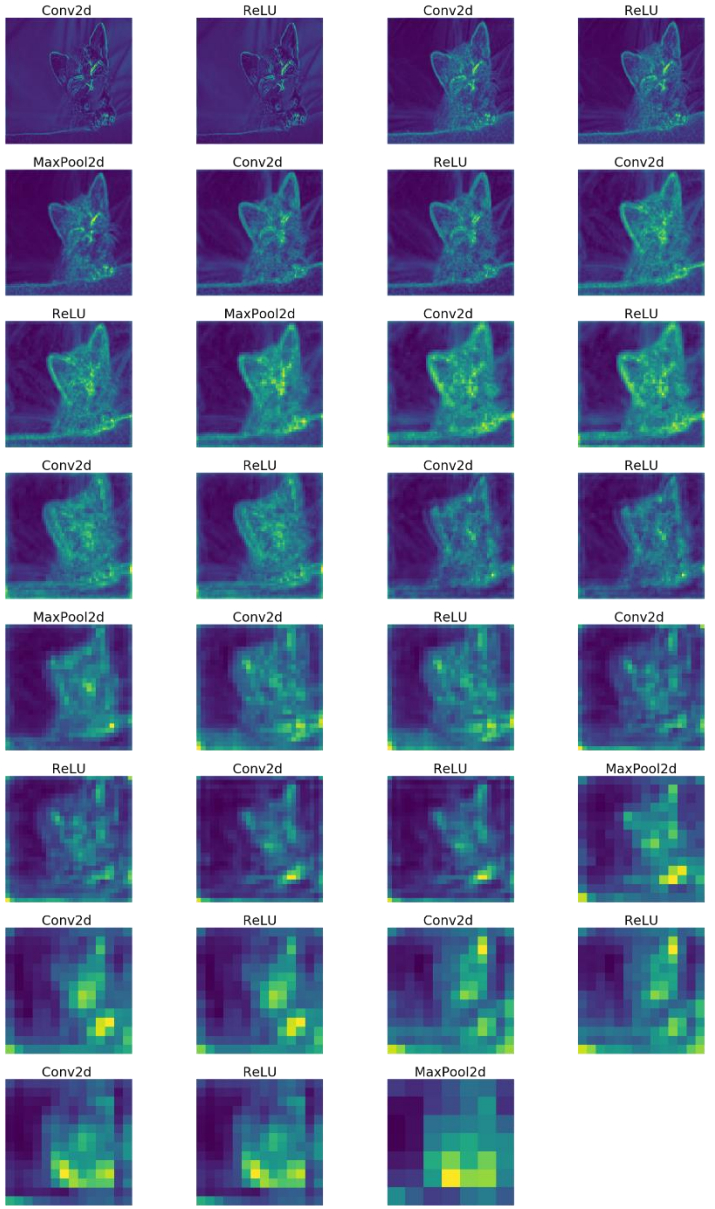


Conv blocks



Classic ConvNet Architecture

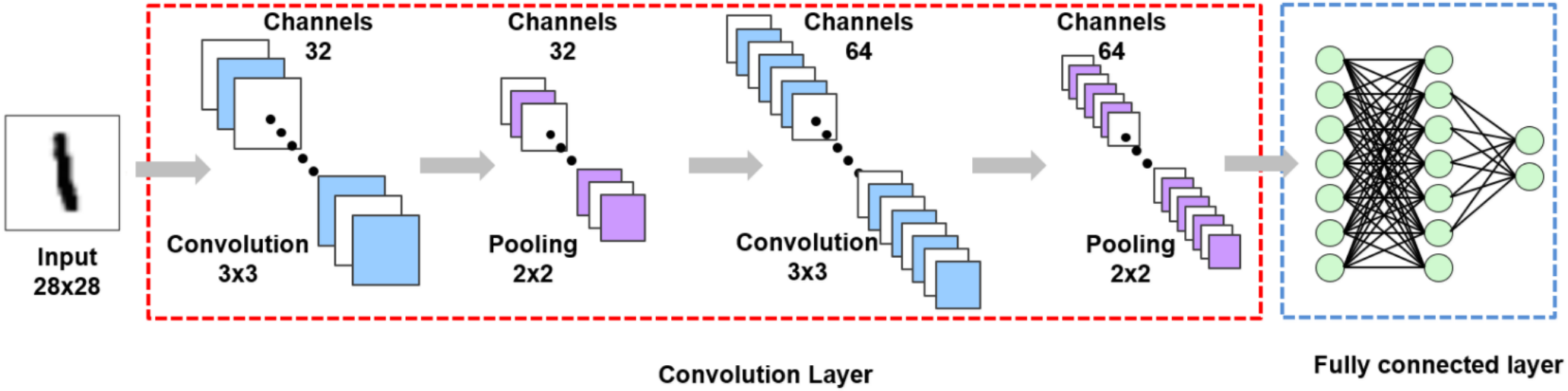
- Input
- Conv blocks
  - Convolution + activation (relu)
  - Convolution + activation (relu)
  - …
  - Maxpooling
- Output
  - Fully connected layers
  - Softmax



CNN in TensorFlow

Lab: CNN with TensorFlow

- MNIST example
- To classify handwritten digits



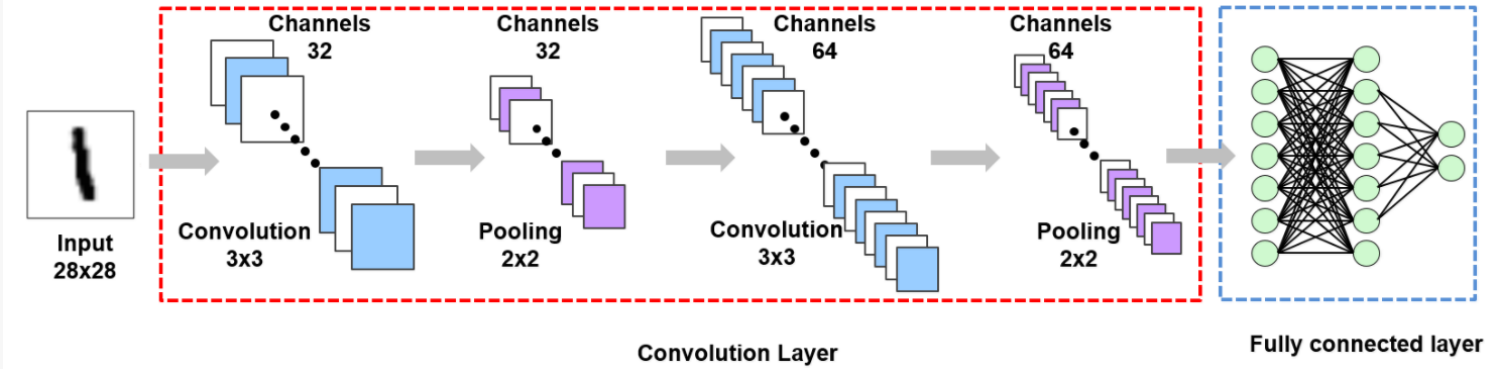
CNN Structure

```
# input layer
input_h = 28 # Input height
input_w = 28 # Input width
input_ch = 1 # Input channel : Gray scale
# (None, 28, 28, 1)

# First convolution layer
k1_h = 3
k1_w = 3
k1_ch = 32
p1_h = 2
p1_w = 2
# (None, 14, 14 ,32)

# Second convolution layer
k2_h = 3
k2_w = 3
k2_ch = 64
p2_h = 2
p2_w = 2
# (None, 7, 7 ,64)

## Fully connected
# Flatten the features -> (None, 7*7*64)
conv_result_size = int((28/(2*2)) * (28/(2*2)) * k2_ch)
n_hidden = 100
n_output = 10
```

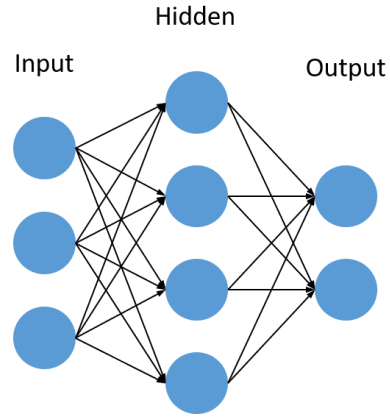


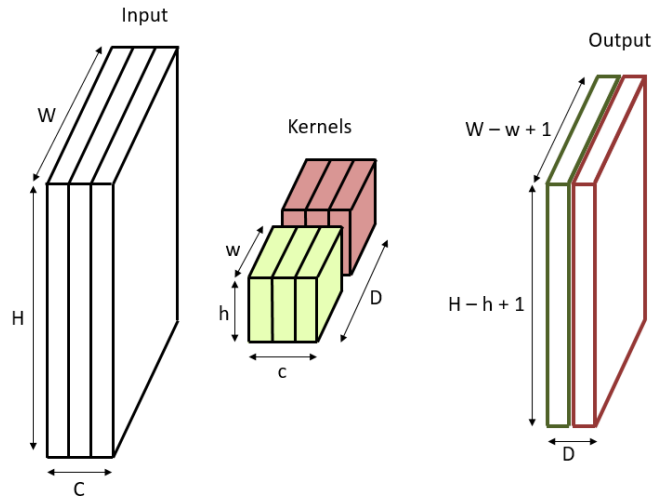
Weights, Biases and Placeholder

```python
# kernel size: [kernel_height, kernel_width, input_ch, output_ch]
weights = {
    'conv1' : tf.Variable(tf.random_normal([k1_h, k1_w, input_ch, k1_ch], stddev = 0.1)),
    'conv2' : tf.Variable(tf.random_normal([k2_h, k2_w, k1_ch, k2_ch], stddev = 0.1)),
    'hidden' : tf.Variable(tf.random_normal([conv_result_size, n_hidden], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_hidden, n_output], stddev = 0.1))
}

# bias size: [output_ch] or [neuron_size]
biases = {
    'conv1' : tf.Variable(tf.random_normal([k1_ch], stddev = 0.1)),
    'conv2' : tf.Variable(tf.random_normal([k2_ch], stddev = 0.1)),
    'hidden' : tf.Variable(tf.random_normal([n_hidden], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_output], stddev = 0.1))
}

# input layer: [batch_size, image_height, image_width, channels]
# output layer: [batch_size, class_size]
x = tf.placeholder(tf.float32, [None, input_h, input_w, input_ch])
y = tf.placeholder(tf.float32, [None, n_output])
```

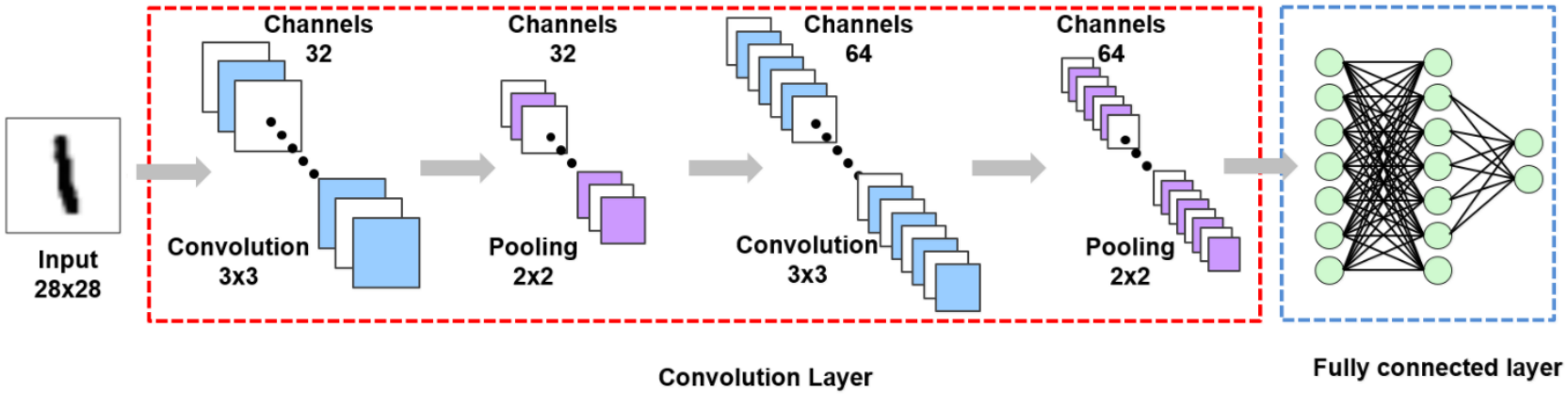




Build a Model

- Convolution layers
  1) The layer performs several convolutions to produce a set of linear activations
  2) Each linear activation is running through a nonlinear activation function
  3) Use pooling to modify the output of the layer further

- Fully connected layers
  - Simple multi-layer perceptrons (MLP)



Convolution

- First, the layer performs several convolutions to produce a set of linear activations
  - Filter size : 3×3
  - Stride : The stride of the sliding window for each dimension of input
  - Padding : Allow us to control the kernel width and the size of the output independently
    - 'SAME' : zero padding
    - 'VALID' : No padding

```
# [batch, height, width, channels]

def net(x, weights, biases):
    # First convolution layer
    conv1 = tf.nn.conv2d(x,
                         weights['conv1'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

    # Second convolution layer
    conv2 = tf.nn.conv2d(maxp1,
                         weights['conv2'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    maxp2_flatten = tf.reshape(maxp2, [-1, conv_result_size])

    # Fully connected
    hidden = tf.add(tf.matmul(maxp2_flatten, weights['hidden']), biases['hidden'])
    hidden = tf.nn.relu(hidden)
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Activation

- Second, each linear activation is running through a nonlinear activation function

```
# [batch, height, width, channels]

def net(x, weights, biases):
    # First convolution layer
    conv1 = tf.nn.conv2d(x,
                         weights['conv1'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

    # Second convolution layer
    conv2 = tf.nn.conv2d(maxp1,
                         weights['conv2'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    maxp2_flatten = tf.reshape(maxp2, [-1, conv_result_size])

    # Fully connected
    hidden = tf.add(tf.matmul(maxp2_flatten, weights['hidden']), biases['hidden'])
    hidden = tf.nn.relu(hidden)
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Pooling

- Third, use a pooling to modify the output of the layer further
  - Compute a maximum value in a sliding window (max pooling)
  - Pooling size : 2×2

```
# [batch, height, width, channels]

def net(x, weights, biases):
    # First convolution layer
    conv1 = tf.nn.conv2d(x,
                         weights['conv1'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

    # Second convolution layer
    conv2 = tf.nn.conv2d(maxp1,
                         weights['conv2'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    maxp2_flatten = tf.reshape(maxp2, [-1, conv_result_size])

    # Fully connected
    hidden = tf.add(tf.matmul(maxp2_flatten, weights['hidden']), biases['hidden'])
    hidden = tf.nn.relu(hidden)
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Second Convolution Layer

```
# [batch, height, width, channels]

def net(x, weights, biases):
    # First convolution layer
    conv1 = tf.nn.conv2d(x,
                         weights['conv1'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

    # Second convolution layer
    conv2 = tf.nn.conv2d(maxp1,
                         weights['conv2'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    maxp2_flatten = tf.reshape(maxp2, [-1, conv_result_size])

    # Fully connected
    hidden = tf.add(tf.matmul(maxp2_flatten, weights['hidden']), biases['hidden'])
    hidden = tf.nn.relu(hidden)
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Fully Connected Layer

- Fully connected layer
  - Input is typically in a form of flattened features
  - Then, apply softmax to multiclass classification problems
  - The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true.

```
# [batch, height, width, channels]

def net(x, weights, biases):
    # First convolution layer
    conv1 = tf.nn.conv2d(x,
                         weights['conv1'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

    # Second convolution layer
    conv2 = tf.nn.conv2d(maxp1,
                         weights['conv2'],
                         strides= [1, 1, 1, 1],
                         padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    maxp2_flatten = tf.reshape(maxp2, [-1, conv_result_size])

    # Fully connected
    hidden = tf.add(tf.matmul(maxp2_flatten, weights['hidden']), biases['hidden'])
    hidden = tf.nn.relu(hidden)
    output = tf.add(tf.matmul(hidden, weights['output']), biases['output'])

    return output
```

Loss and Optimizer

- Loss
  - Classification: Cross entropy
  - Equivalent to applying logistic regression
- Optimizer
  - GradientDescentOptimizer
  - AdamOptimizer: the most popular optimizer

$$-\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_\theta\left(x^{(i)}\right)) + (1-y^{(i)}) \log(1-h_\theta\left(x^{(i)}\right))$$

```
LR = 0.0001

pred = net(x, weights, biases)
loss = tf.nn.softmax_cross_entropy_with_logits(labels = y, logits = pred)
loss = tf.reduce_mean(loss)

optm = tf.train.AdamOptimizer(LR).minimize(loss)
```

Optimization

```
n_batch = 50
n_iter = 2500
n_prt = 250

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

loss_record_train = []
loss_record_test = []
for epoch in range(n_iter):
    train_x, train_y = mnist.train.next_batch(n_batch)
    train_x = np.reshape(train_x, [-1, input_h, input_w, input_ch])
    sess.run(optm, feed_dict = {x: train_x,  y: train_y})

    if epoch % n_prt == 0:
        test_x, test_y = mnist.test.next_batch(n_batch)
        test_x = np.reshape(test_x, [-1, input_h, input_w, input_ch])
        c1 = sess.run(loss, feed_dict = {x: train_x, y: train_y})
        c2 = sess.run(loss, feed_dict = {x: test_x, y: test_y})
        loss_record_train.append(c1)
        loss_record_test.append(c2)
        print ("Iter : {}".format(epoch))
        print ("Cost : {}".format(c1))
```

Test or Evaluation

```
test_x, test_y = mnist.test.next_batch(100)

my_pred = sess.run(pred, feed_dict = {x: test_x.reshape(-1, 28, 28, 1)})
my_pred = np.argmax(my_pred, axis = 1)

labels = np.argmax(test_y, axis = 1)

accr = np.mean(np.equal(my_pred, labels))
print("Accuracy : {}%".format(accr*100))
```

```
Accuracy : 97.0%
```

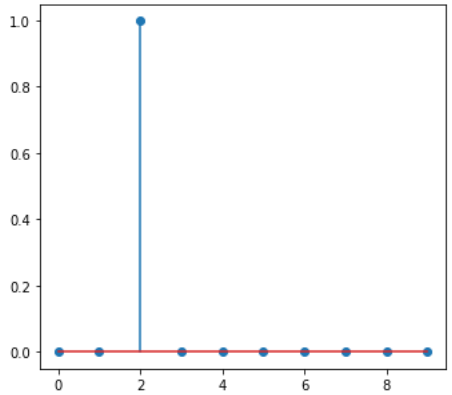
Test or Evaluation

```
test_x, test_y = mnist.test.next_batch(1)
logits = sess.run(tf.nn.softmax(pred), feed_dict = {x: test_x.reshape(-1, 28, 28, 1)})
predict = np.argmax(logits)

plt.figure(figsize = (12,5))
plt.subplot(1,2,1)
plt.imshow(test_x.reshape(28, 28), 'gray')
plt.axis('off')
plt.subplot(1,2,2)
plt.stem(logits.ravel())
plt.show()

np.set_printoptions(precision = 2, suppress = True)
print('Prediction : {}'.format(predict))
print('Probability : {}'.format(logits.ravel()))
```

```
Prediction : 2
Probability : [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
```



CNN Implemented in an Embedded System