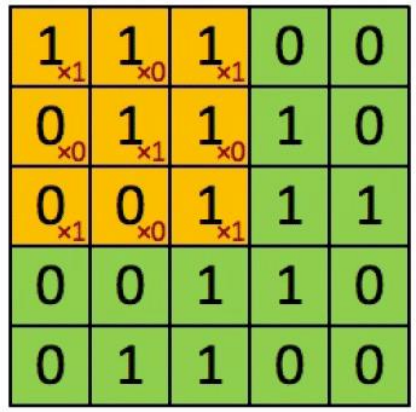
# Convolutional Neural Networks (CNN)

**Industrial AI Lab.**

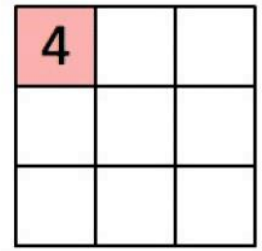
**Prof. Seungchul Lee**

# Convolution on Image

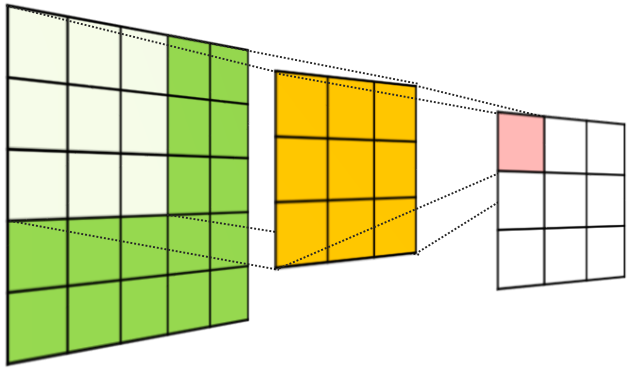
- Convolution in 2D
- Filter (or Kernel)
  - Modify or enhance an image by filtering
  - Filter images to emphasize certain features or remove other features
  - Filtering includes smoothing, sharpening and edge enhancement
  - Discrete convolution can be viewed as **element-wise multiplication** by a matrix




Image



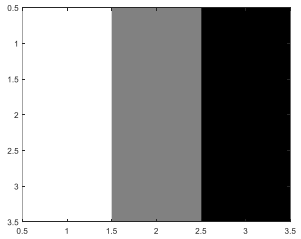

Convolved Feature



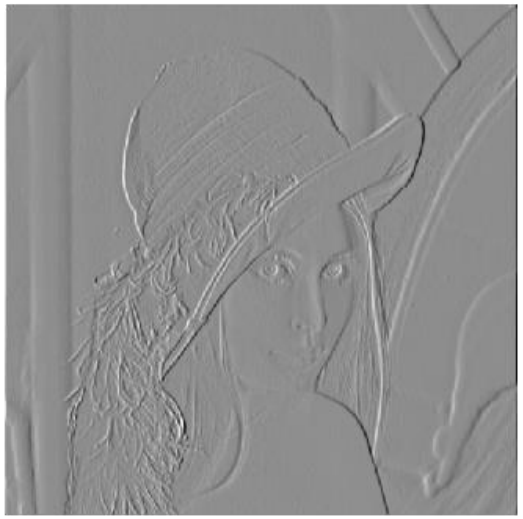
# Convolution on Image

Image

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




Kernel

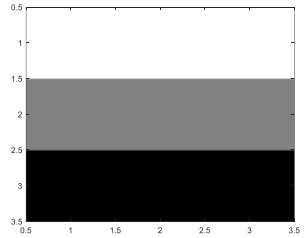


Output

# Convolution on Image

Image

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



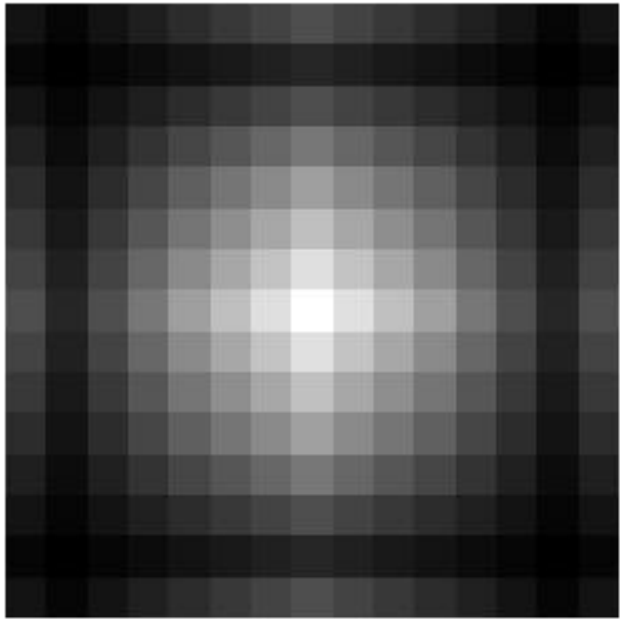
Kernel

Output

# Convolution on Image

Input image

Image filter (15 x 15)



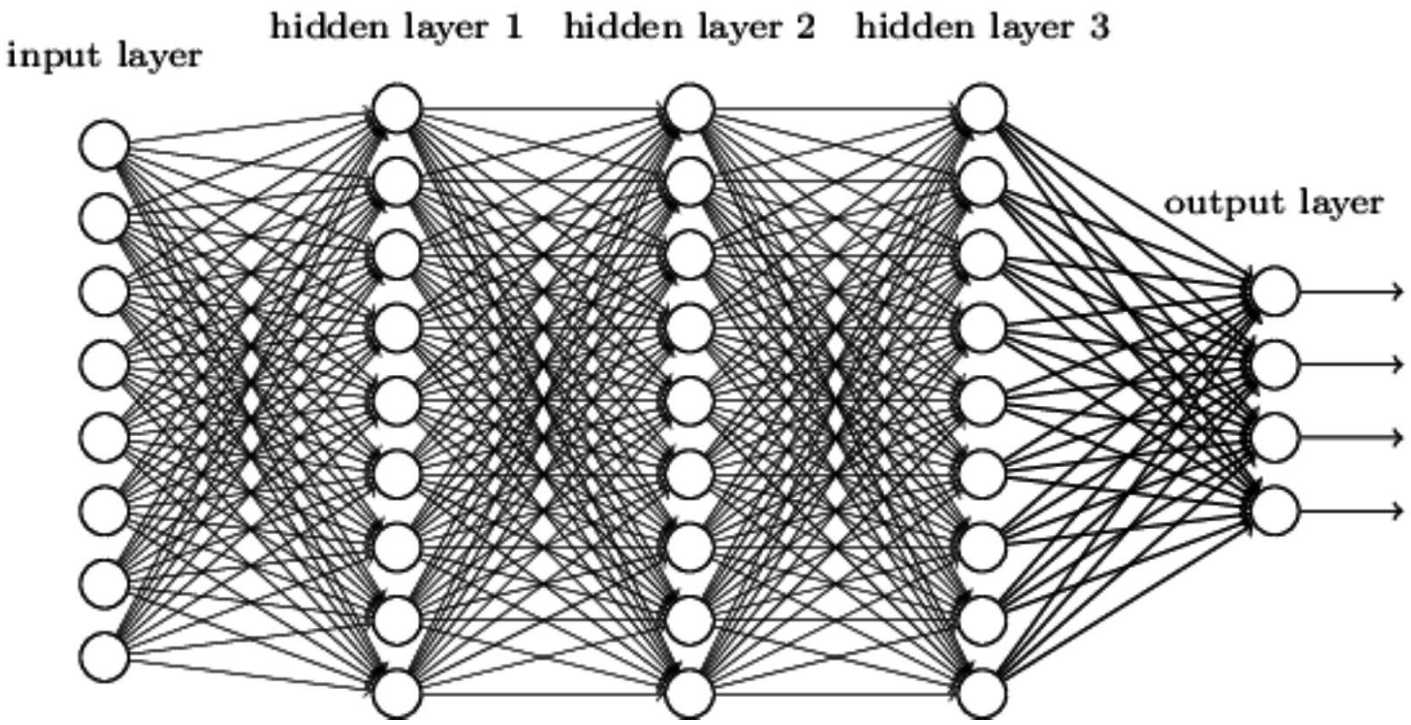
Convoluted image

# 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

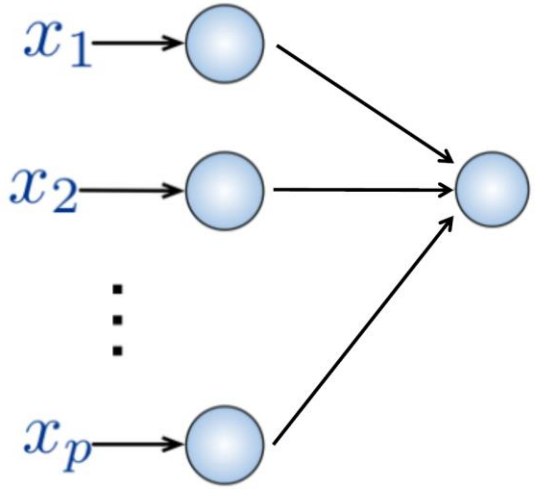
# Learning Visual Features

# Fully Connected Neural Network (= ANN)



# Fully Connected Neural Network

- Input
  - 2D image
  - Vector of pixel values

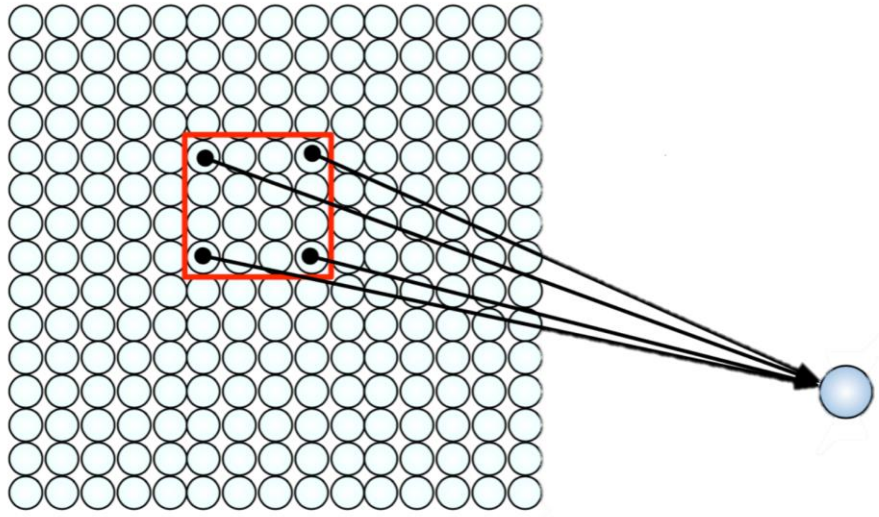



- Fully connected
  - Connect neuron in hidden layer to all neurons in input layer
  - No spatial information
  - Spatial organization of the input is destroyed by flatten
  - And many, many parameters !

- How can we use spatial structure in the input to inform the architecture of the network?

# Using Spatial Structure

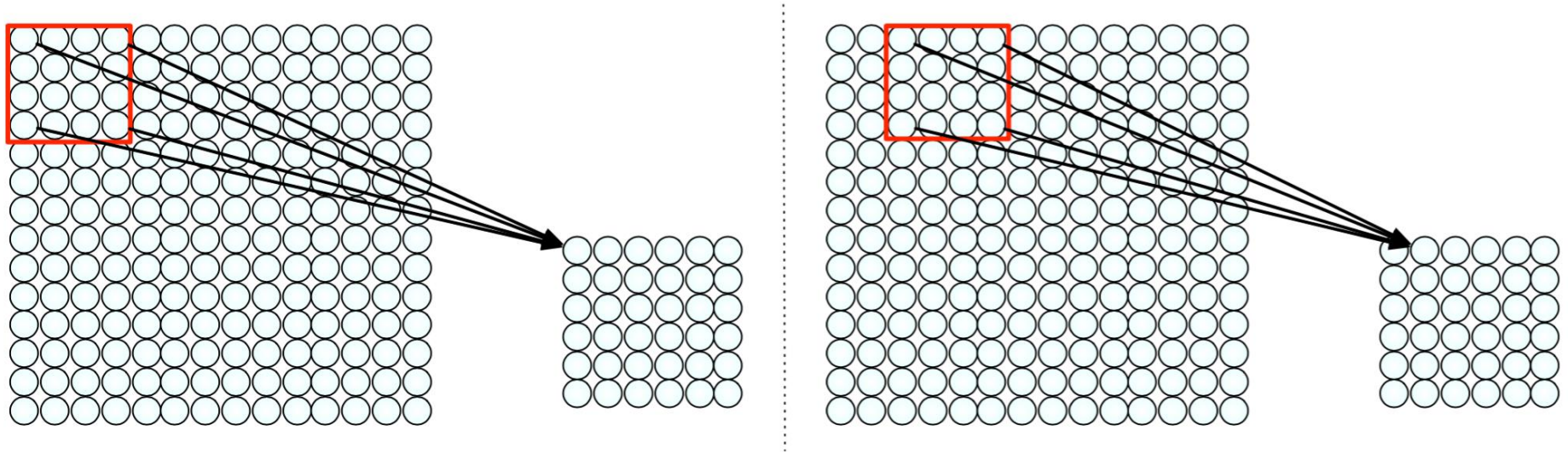
- Input
  - 2D image
  - Matrix of pixel values



- Idea
  - Connect patches of input to neurons in hidden layer
  - Neuron connected to region of input. Only “sees” these values

# Using Spatial Structure

- Connect patch in input layer to a single neuron in subsequent layer.
- Use a sliding window to define connections.
- How can we weight the patch to detect particular features?

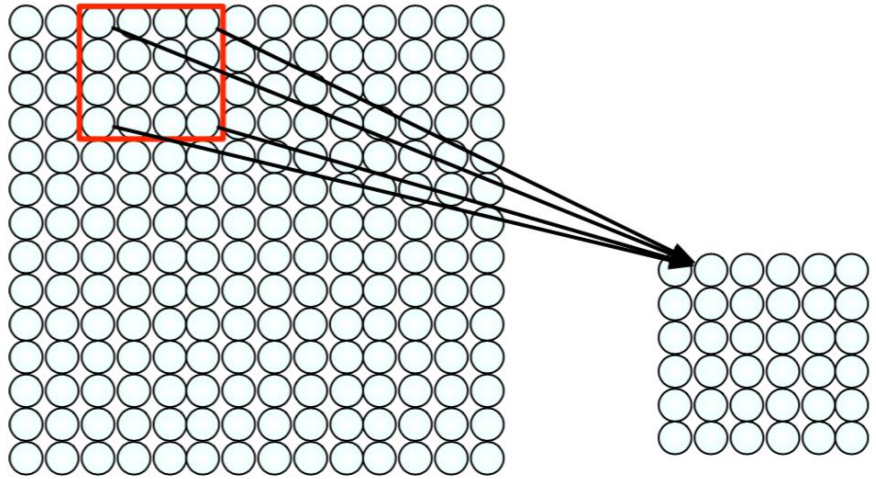


# Applying Filters to Extract Features

1) Apply a set of weights - a filter - to extract local features
2) Use multiple filters to extract different features
3) Spatially share parameters of each filter (features that matter in one part of the input should matter elsewhere)

- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

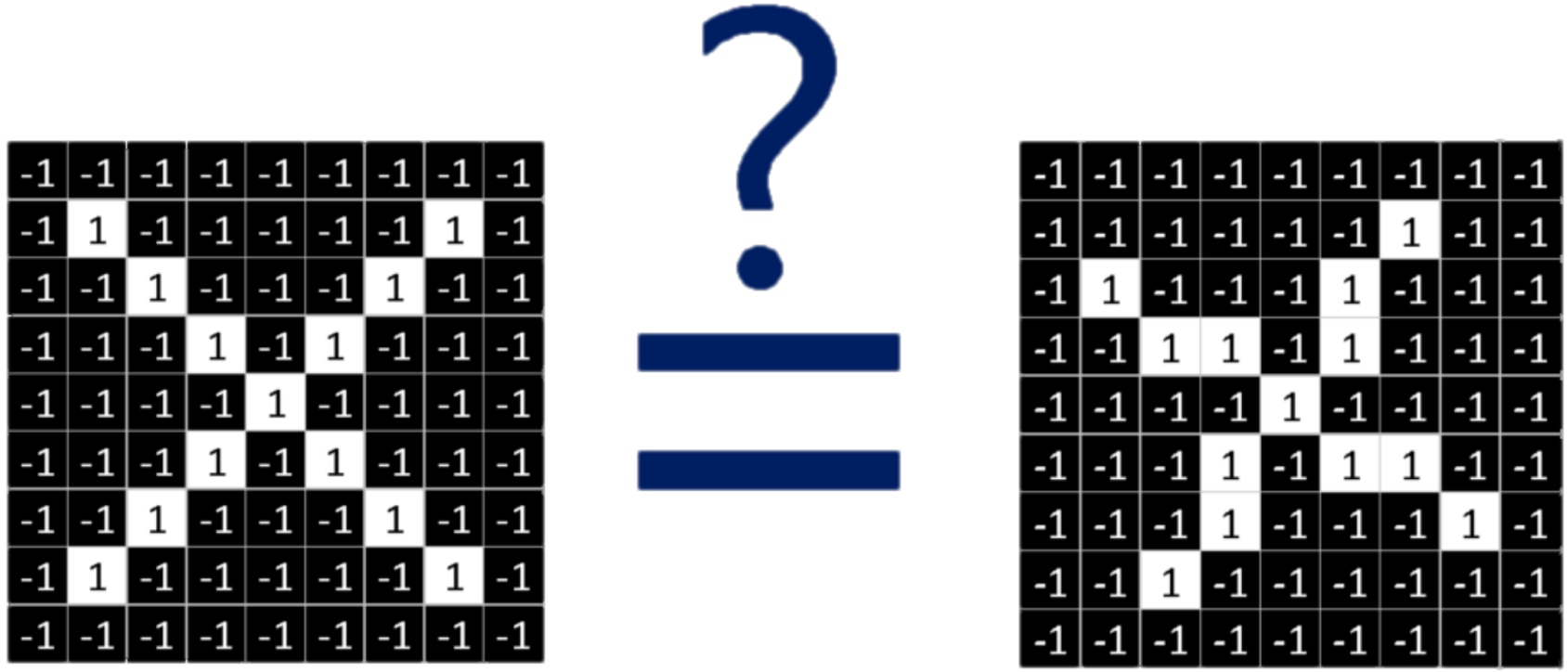
- This “patchy” operation is convolution



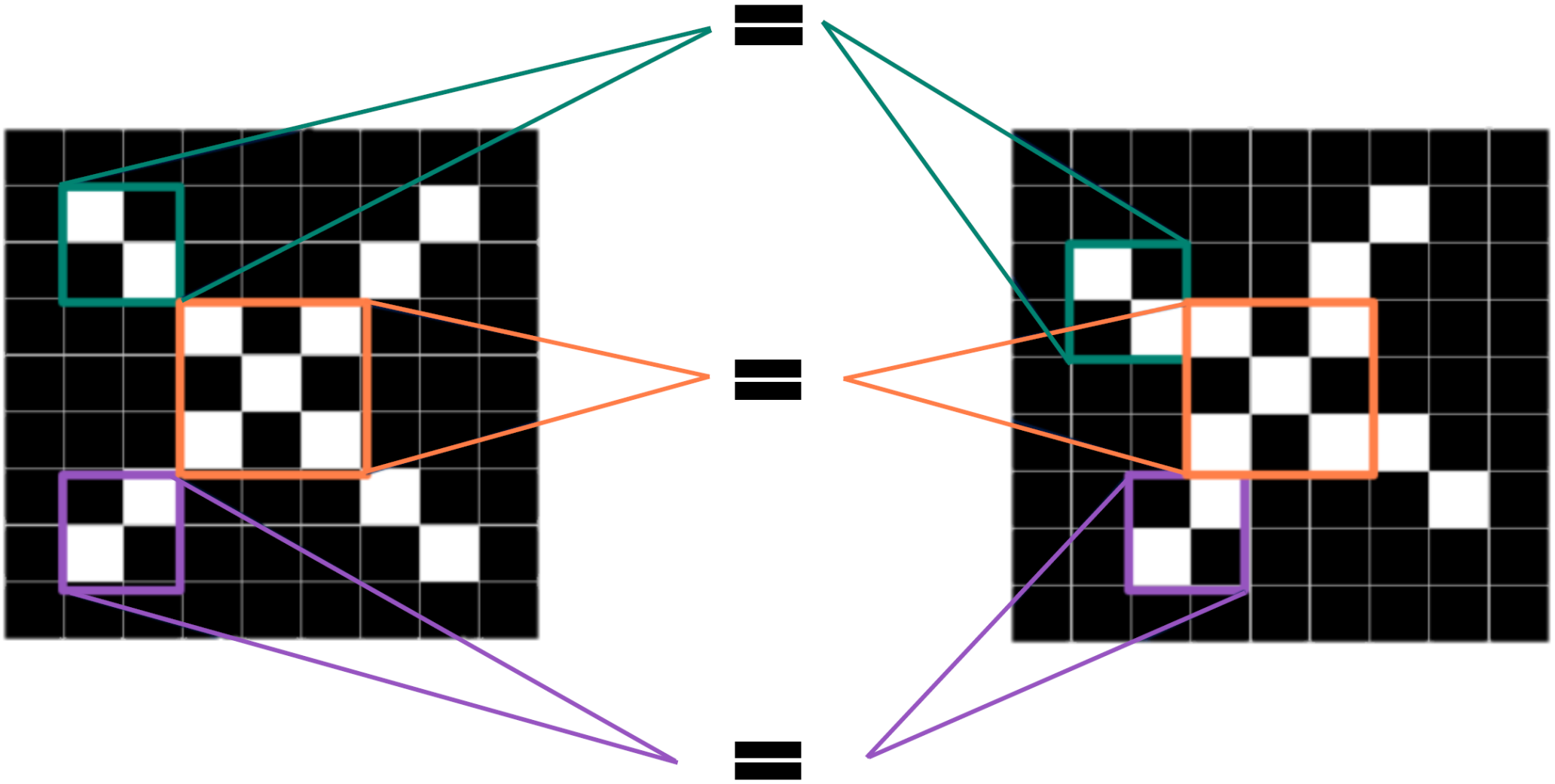
# Feature Extraction and Convolution

# X or X?

- Image is represented as matrix of pixel values…and computers are literal !
- We want to be able to classify an X as an X even if it’s shifted, shrunk, rotated, deformed.

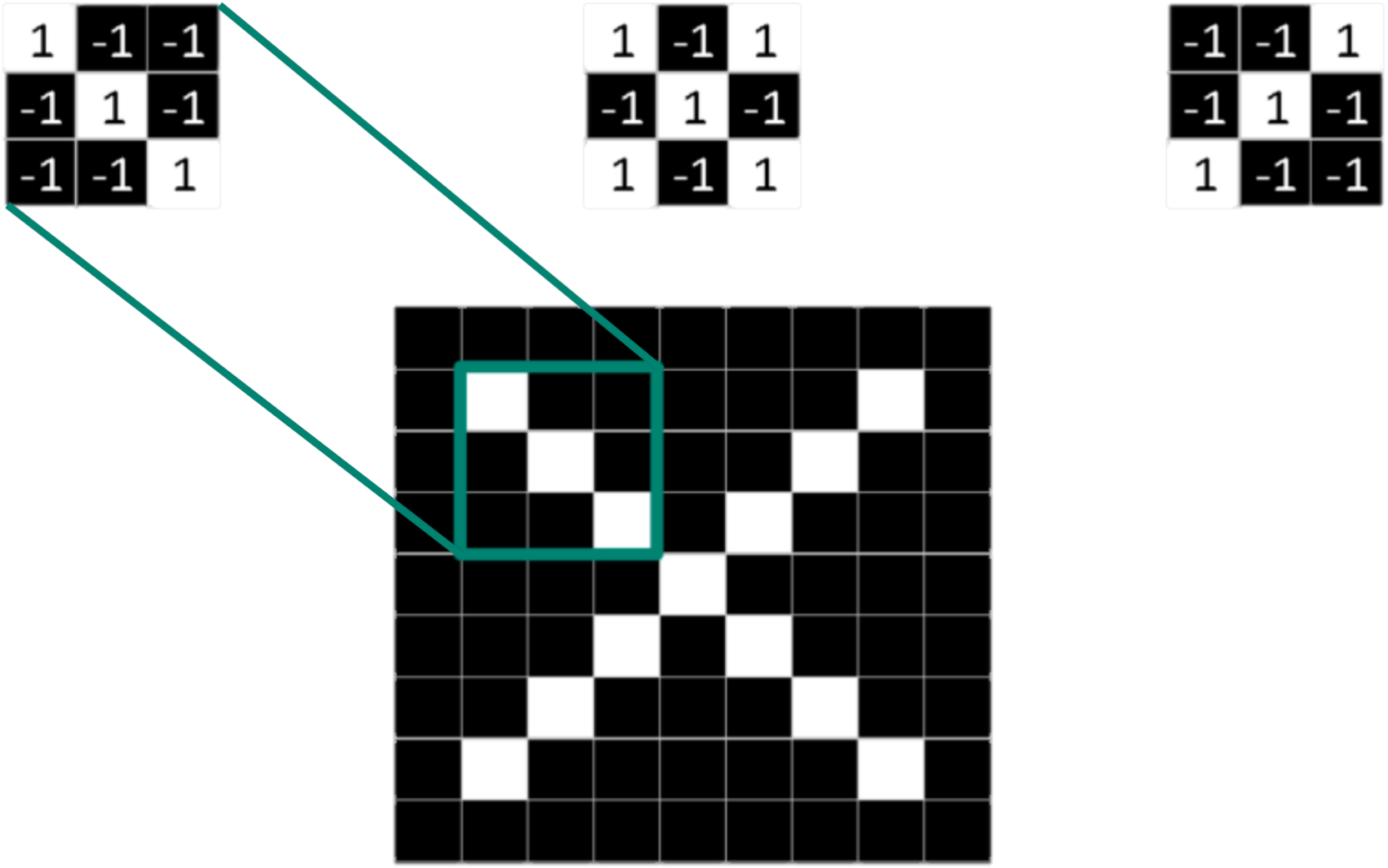


# Features of X

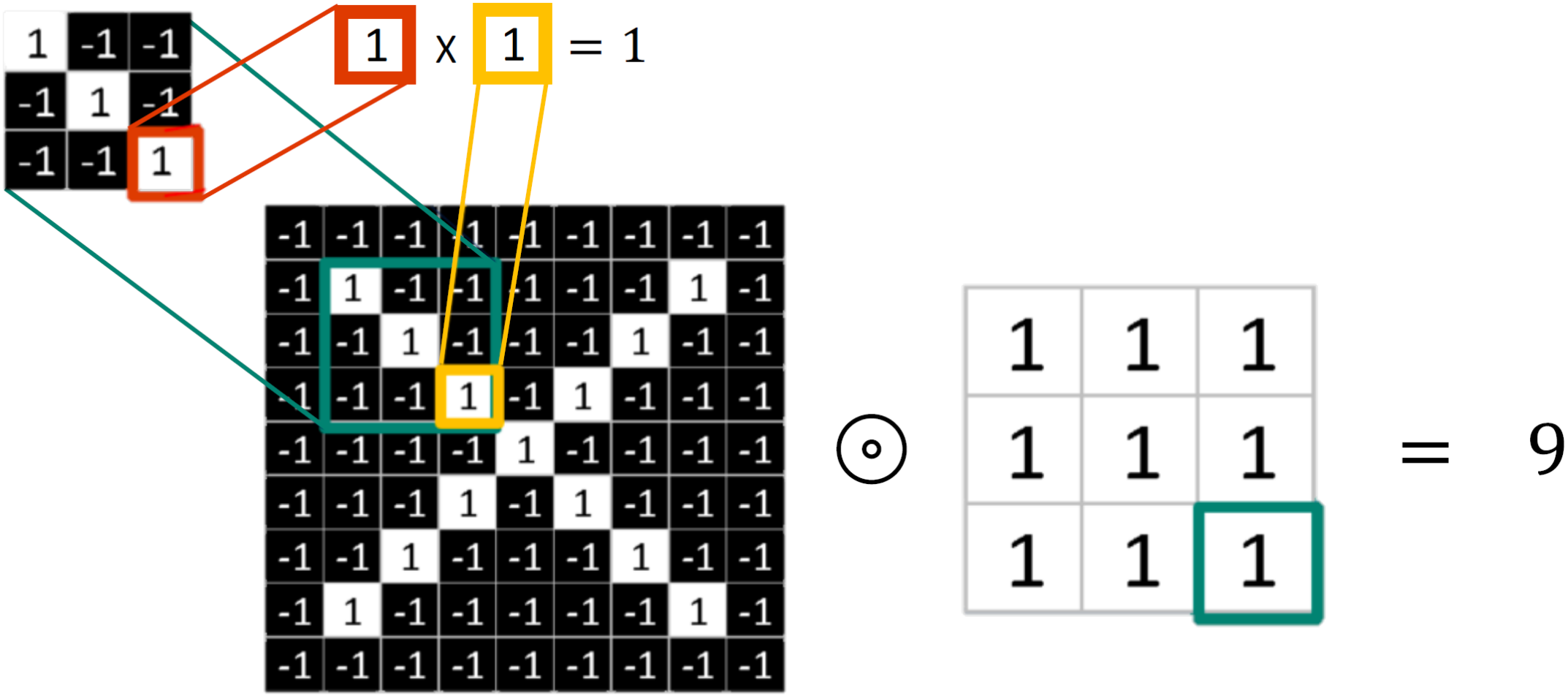


# Filters to Detect X Features

filters



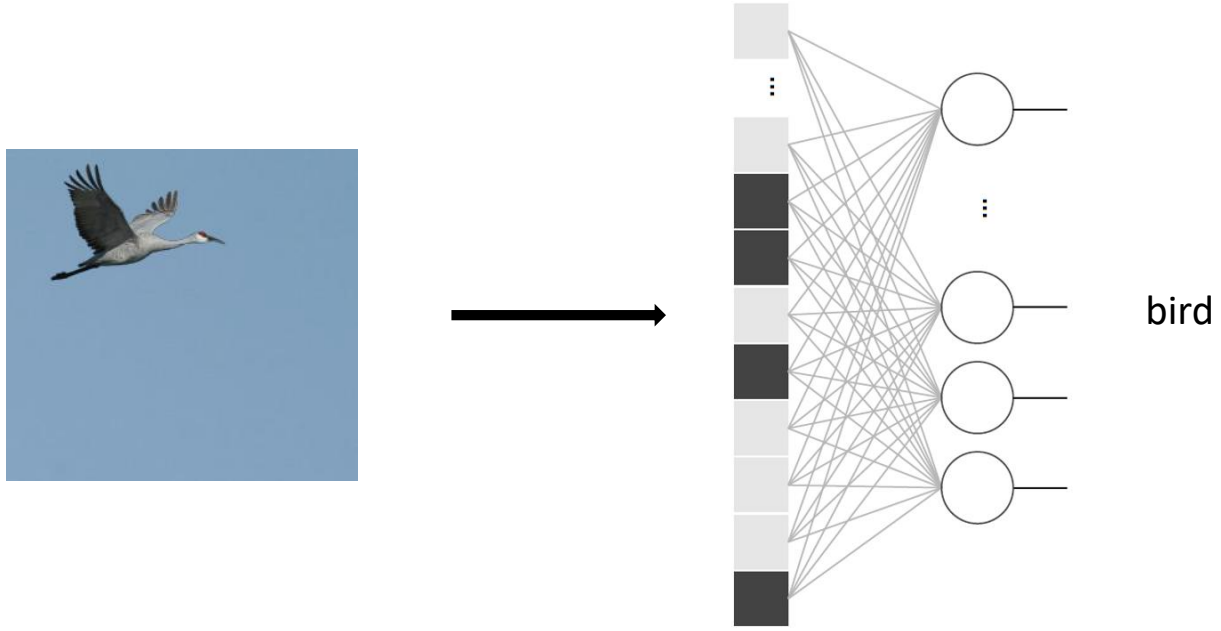
# The Convolution Operation



# Convolutional Neural Networks (CNN)

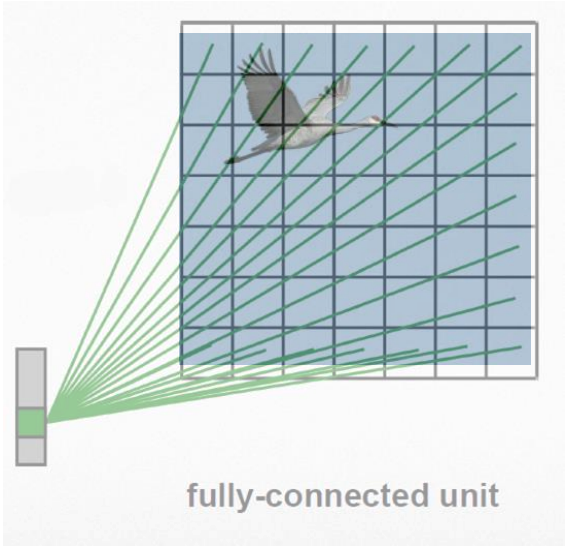
- Motivation
  - The bird occupies a local area and looks the same in different parts of an image. We should construct neural networks which exploit these properties.

# ANN Structure for Object Detection in Image




- Does not seem the best
- Did not make use of the fact that we are dealing with images

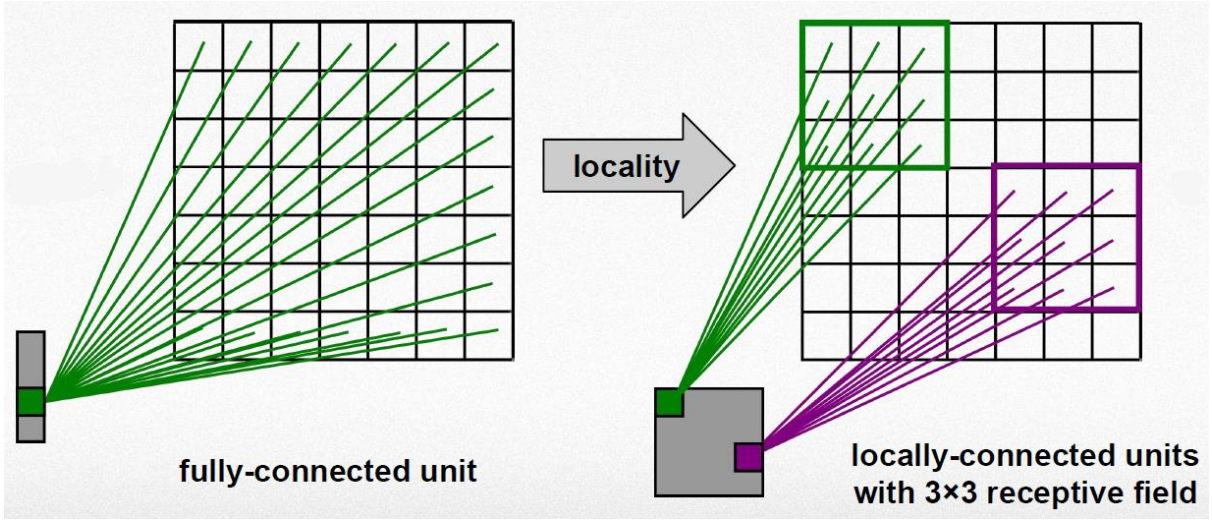
# Convolution Mask + Neural Network




- Does not seem the best
- Did not make use of the fact that we are dealing with images

# Locality

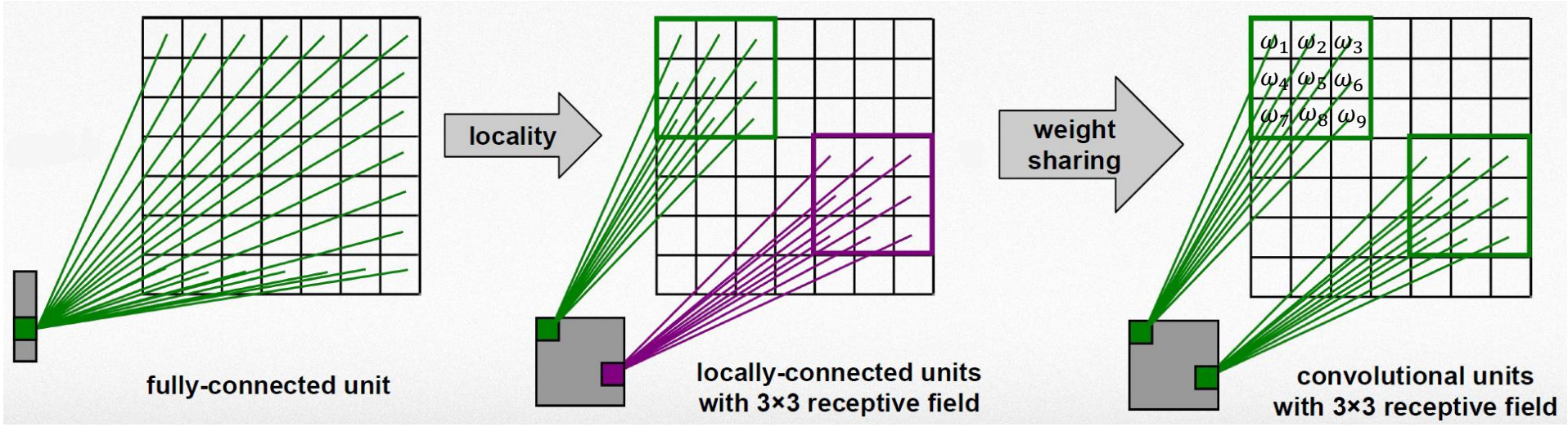
- Locality: objects tend to have a local spatial support
  - fully-connected layer → locally-connected layer



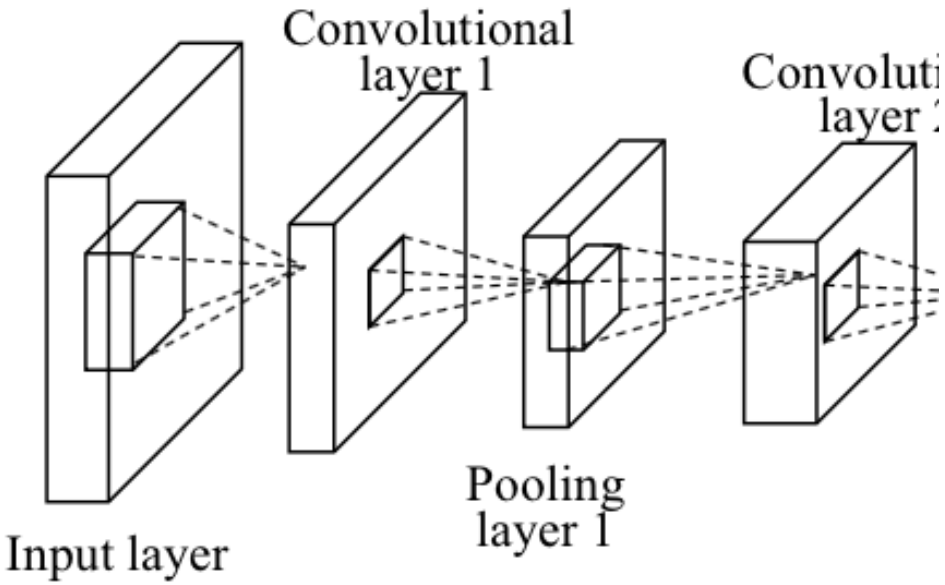
# Locality

- Locality: objects tend to have a local spatial support
  - fully-connected layer → locally-connected layer

We are not designing the kernel, but are learning the kernel from data
→ Learning feature extractor from data

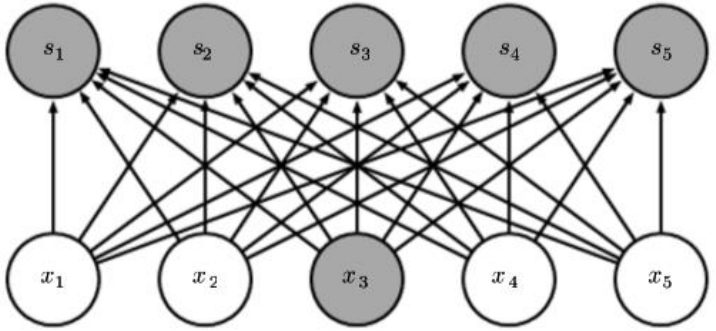


# Object Size?

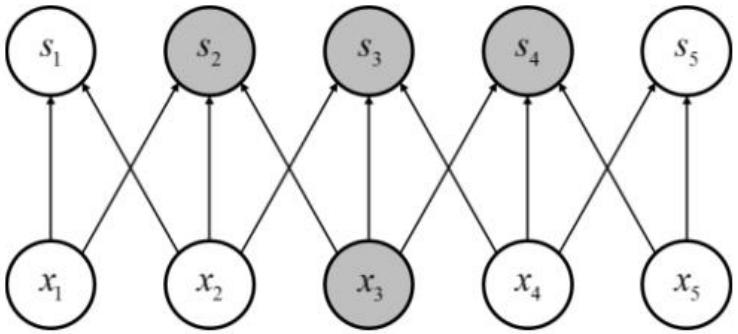

Convolutional
layer 1
Convoluti
layer
Pooling
layer 1
Input layer

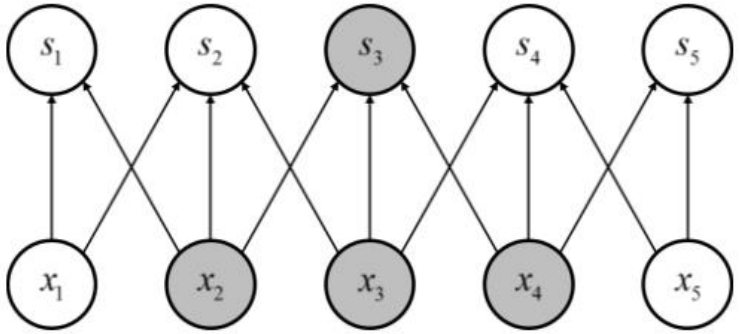
# Convolutional Neural Networks (CNN)

- Matrix multiplication
  - Every output unit interacts with every interacts unit



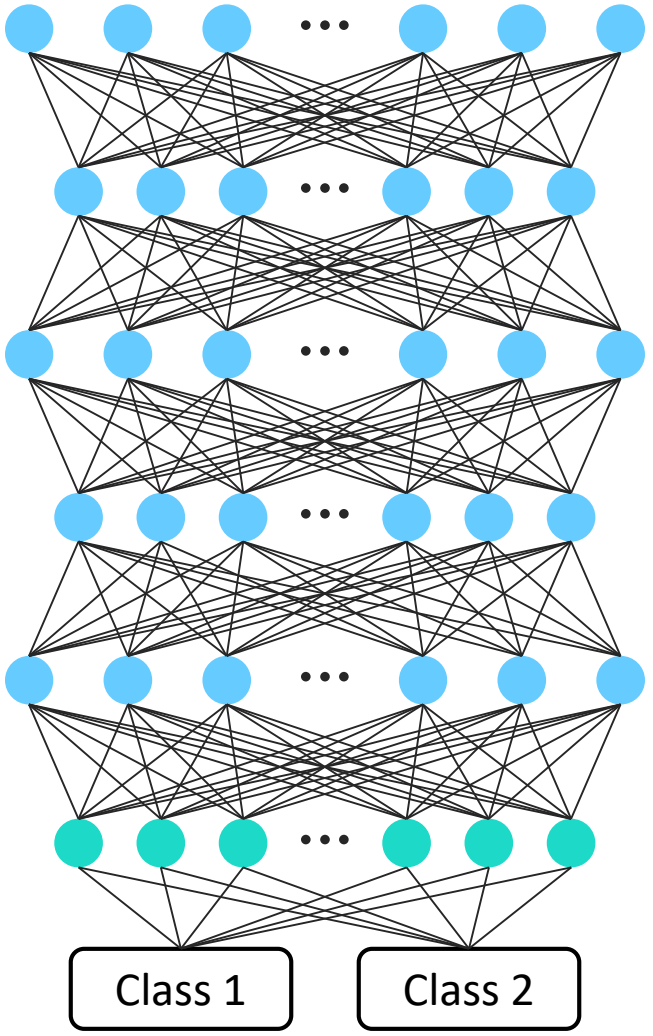

- Convolution
  - Typically have sparse interactions
  - Local connectivity



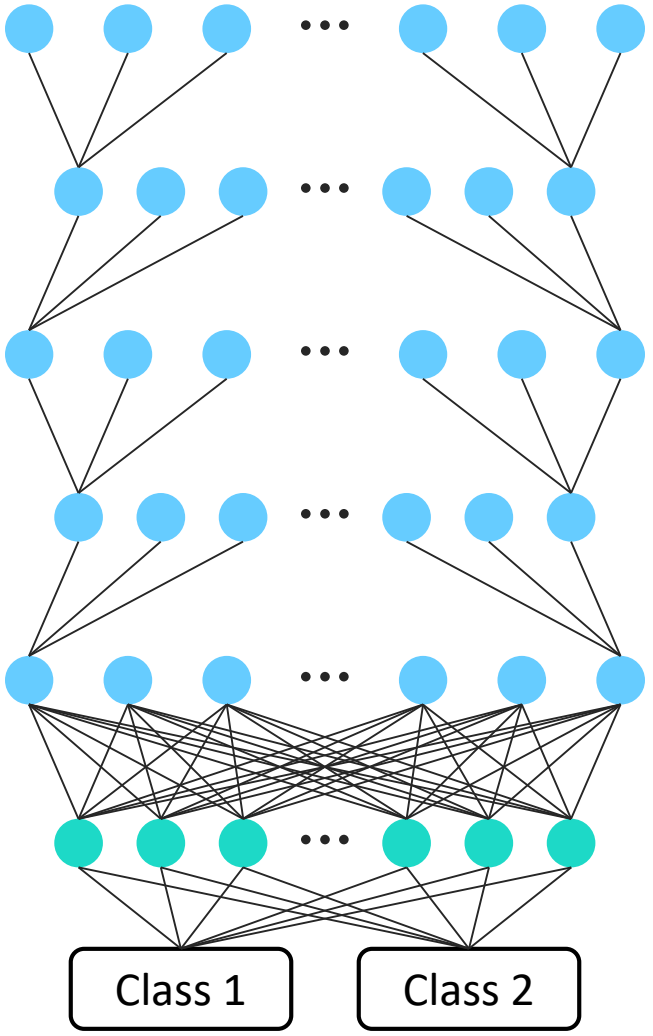
# Deep Artificial Neural Networks

- Universal function approximator
  - Simple nonlinear neurons
  - Linear connected networks
- Hidden layers
  - Autonomous feature learning

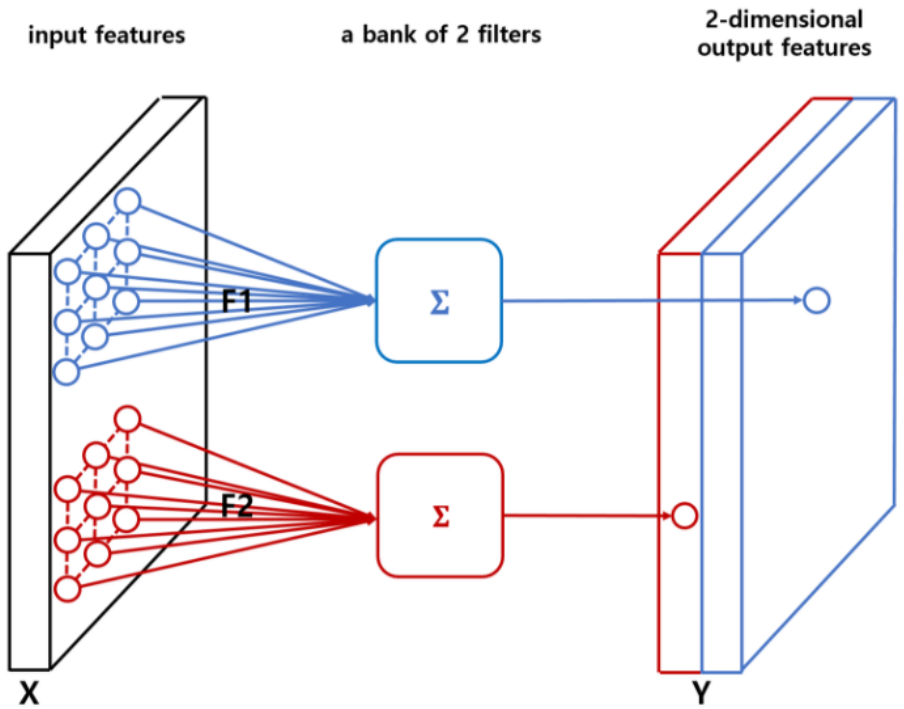


# Convolutional Neural Networks

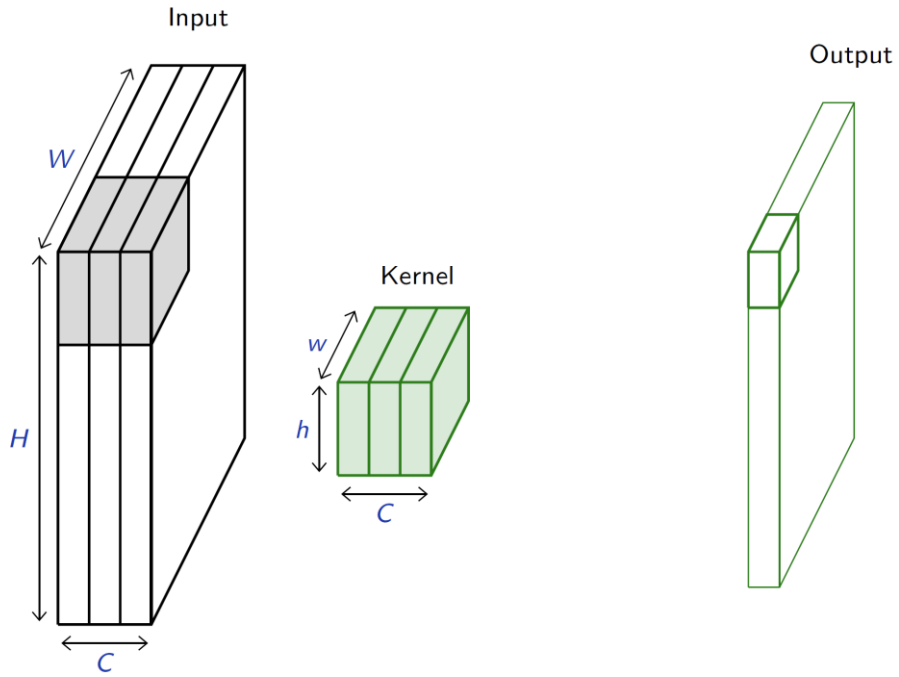
- Structure
  - Weight sharing
  - Local connectivity
- Optimization
  - Smaller searching space



# Multiple Filters (or Kernels)

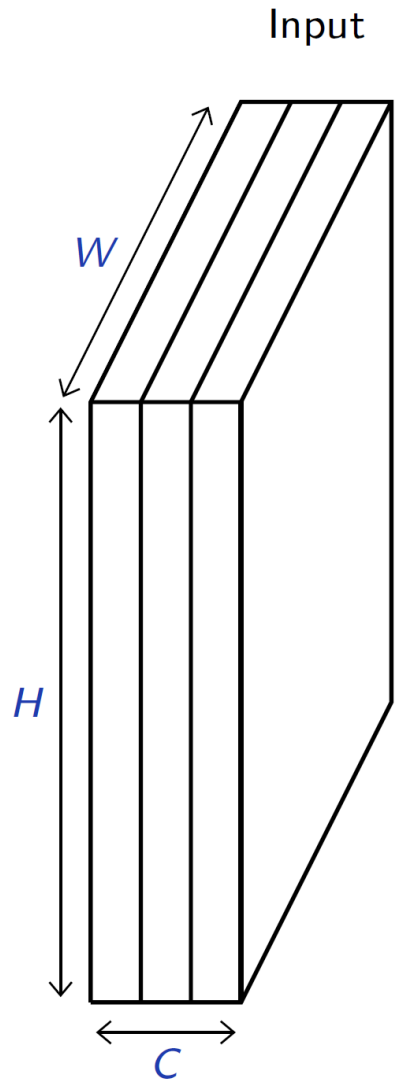


# Channels

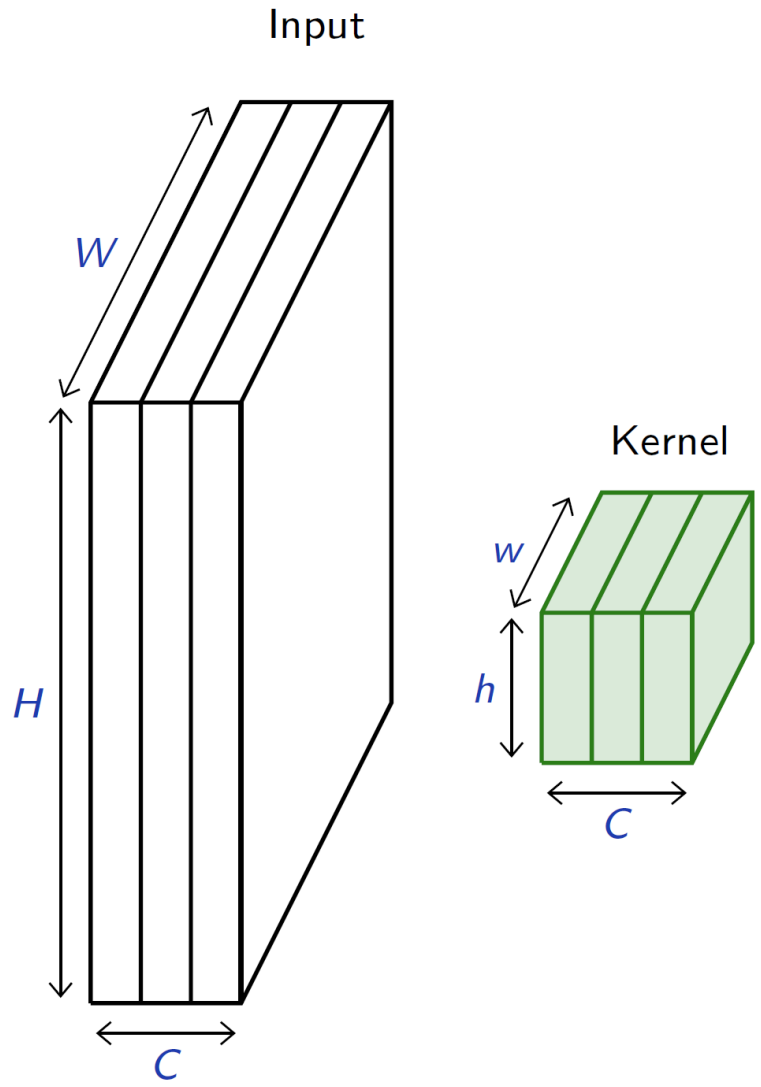



- Colored image = tensor of shape (height, width, channels)
- Convolutions are usually computed for each channel and summed:
- Kernel size aka receptive field (usually 1, 3, 5, 7, 11)

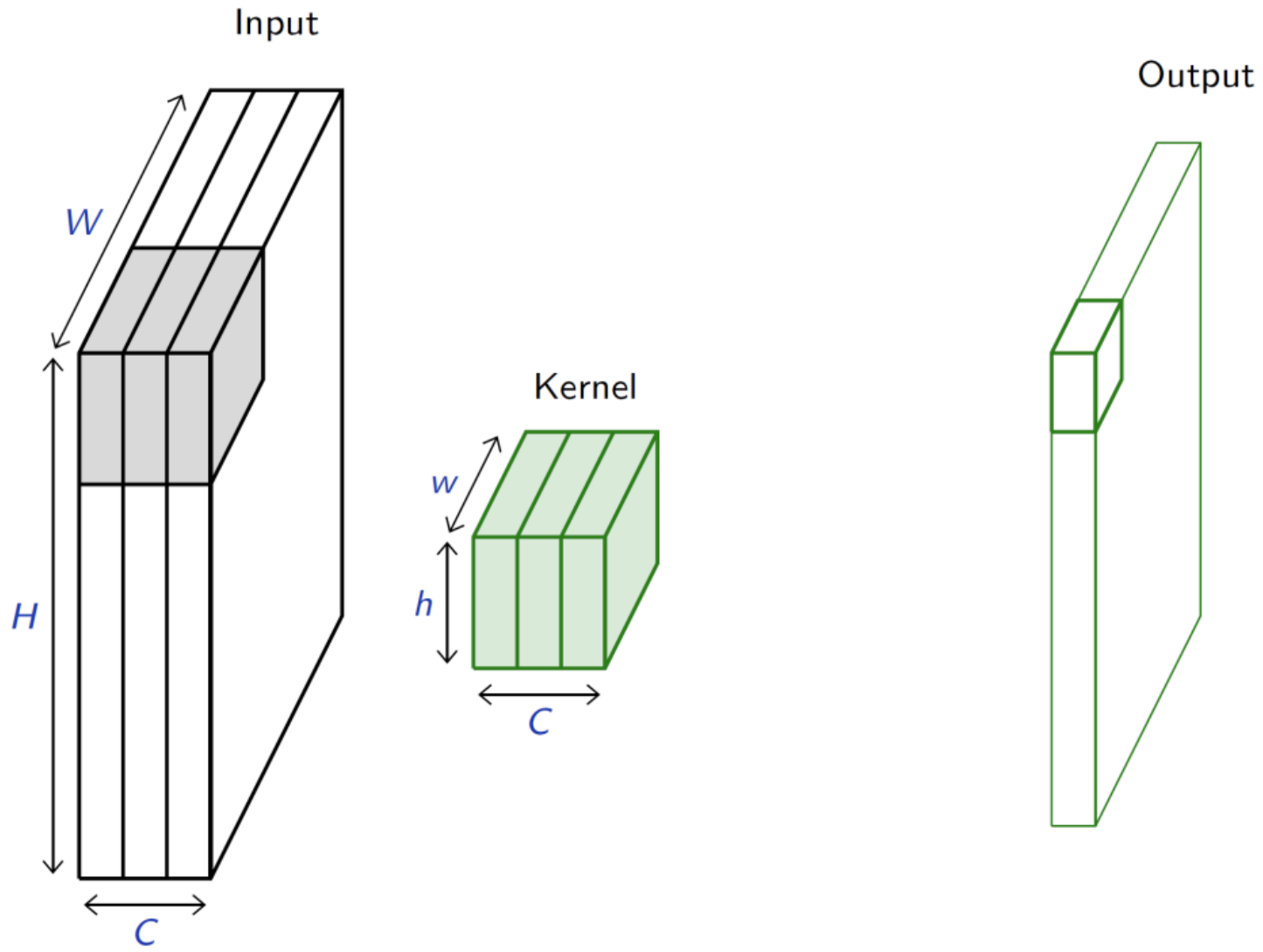
# Multi-channel 2D Convolution



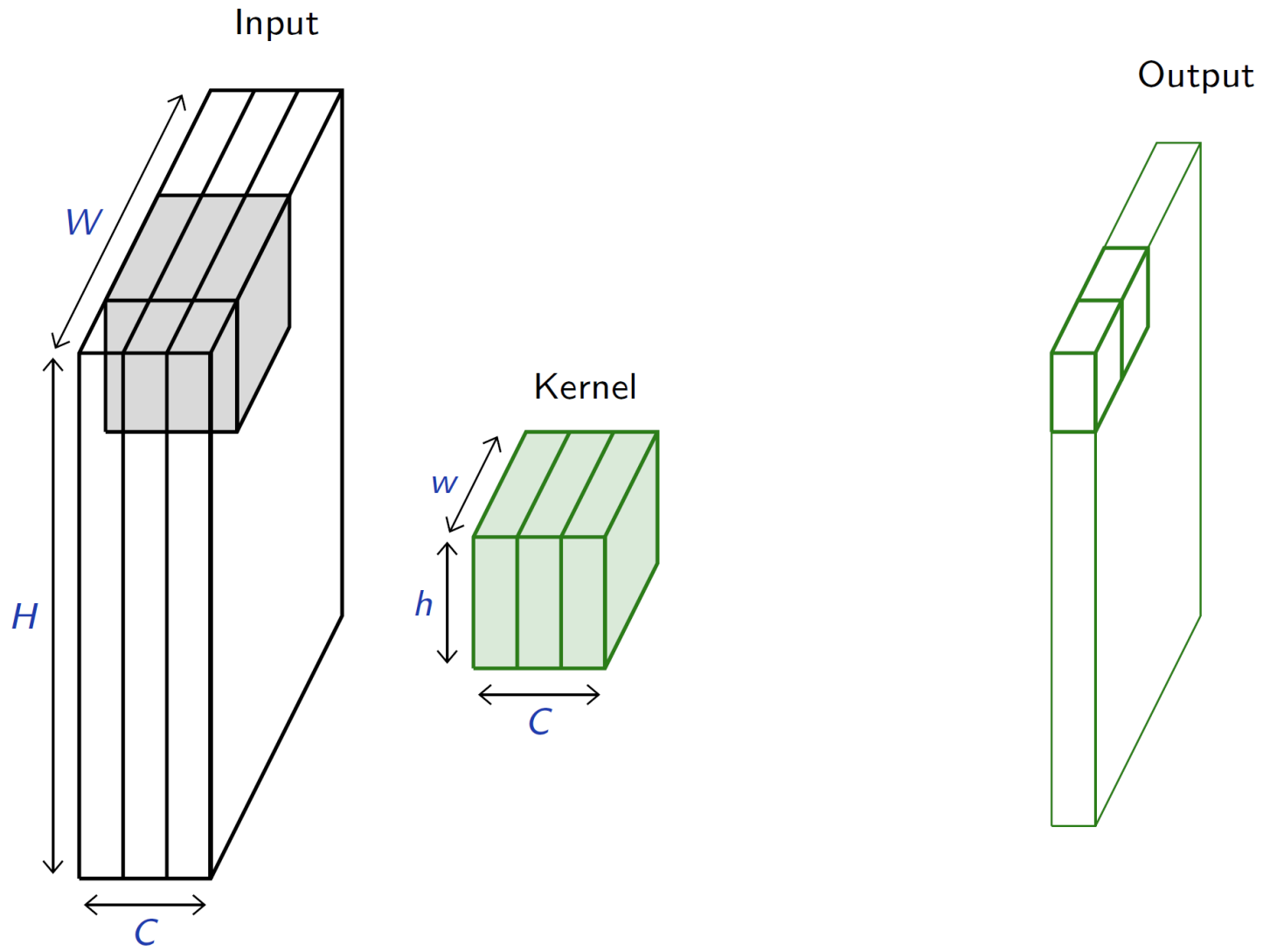
# Multi-channel 2D Convolution



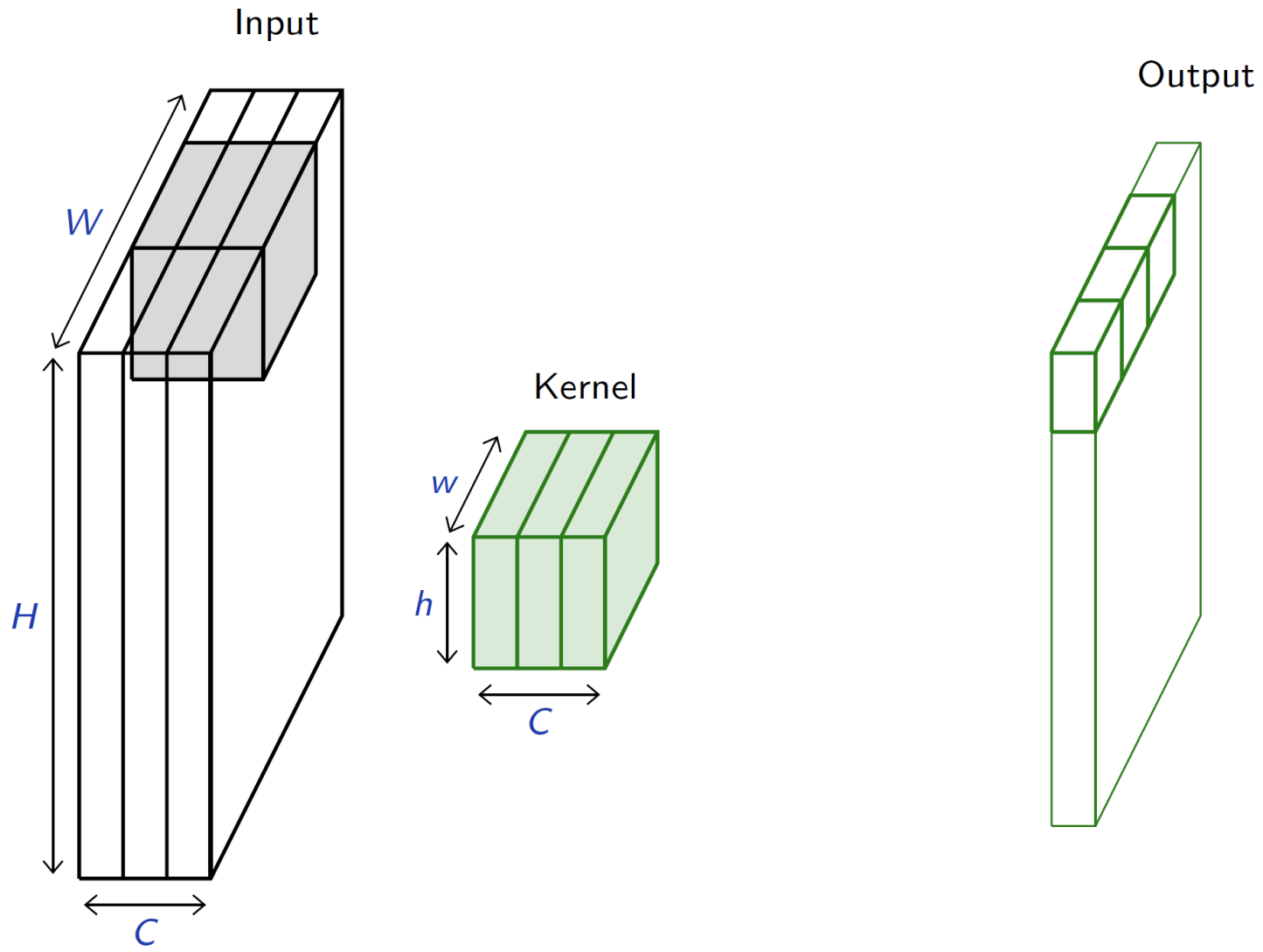
# Multi-channel 2D Convolution



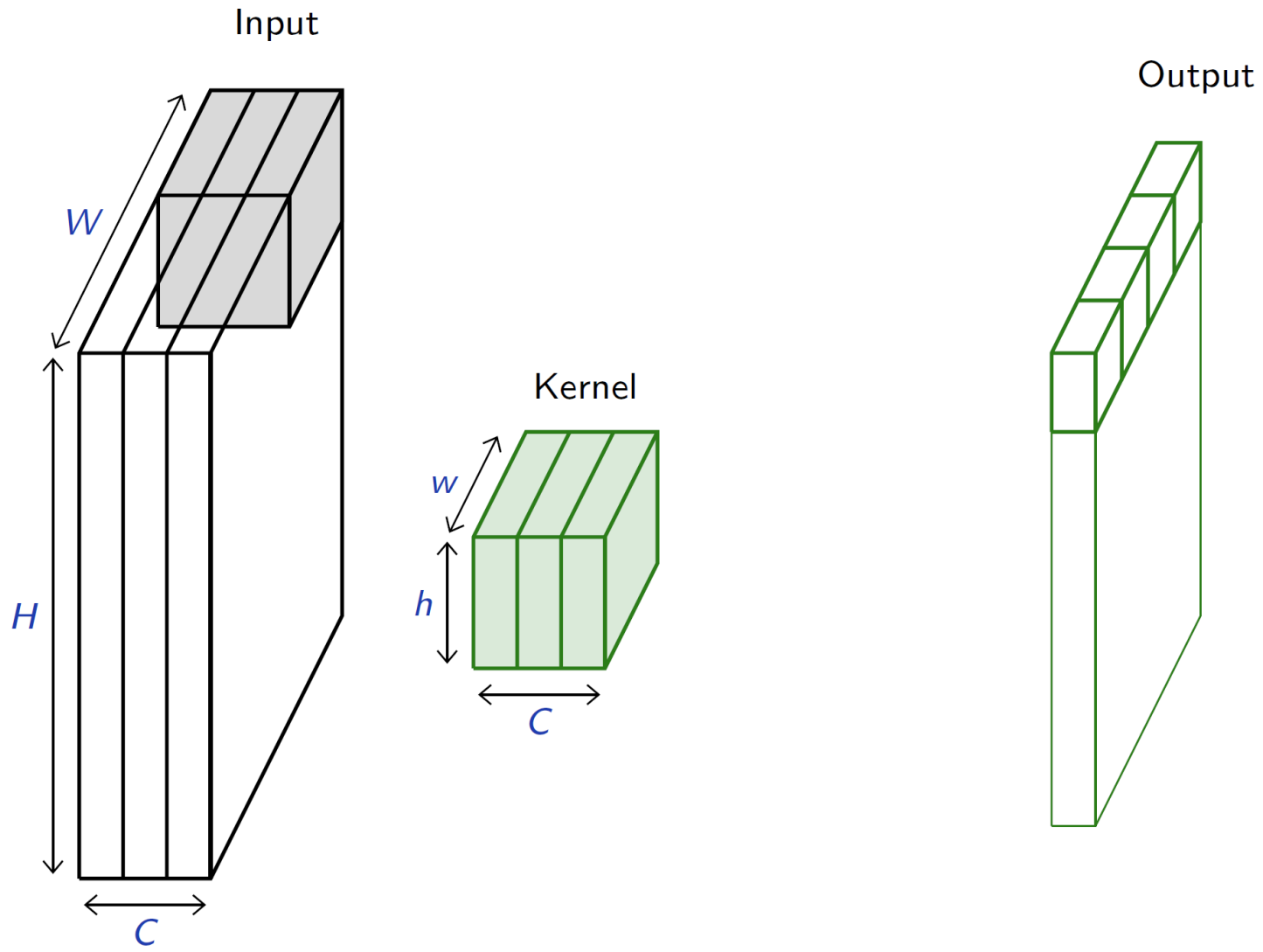
# Multi-channel 2D Convolution



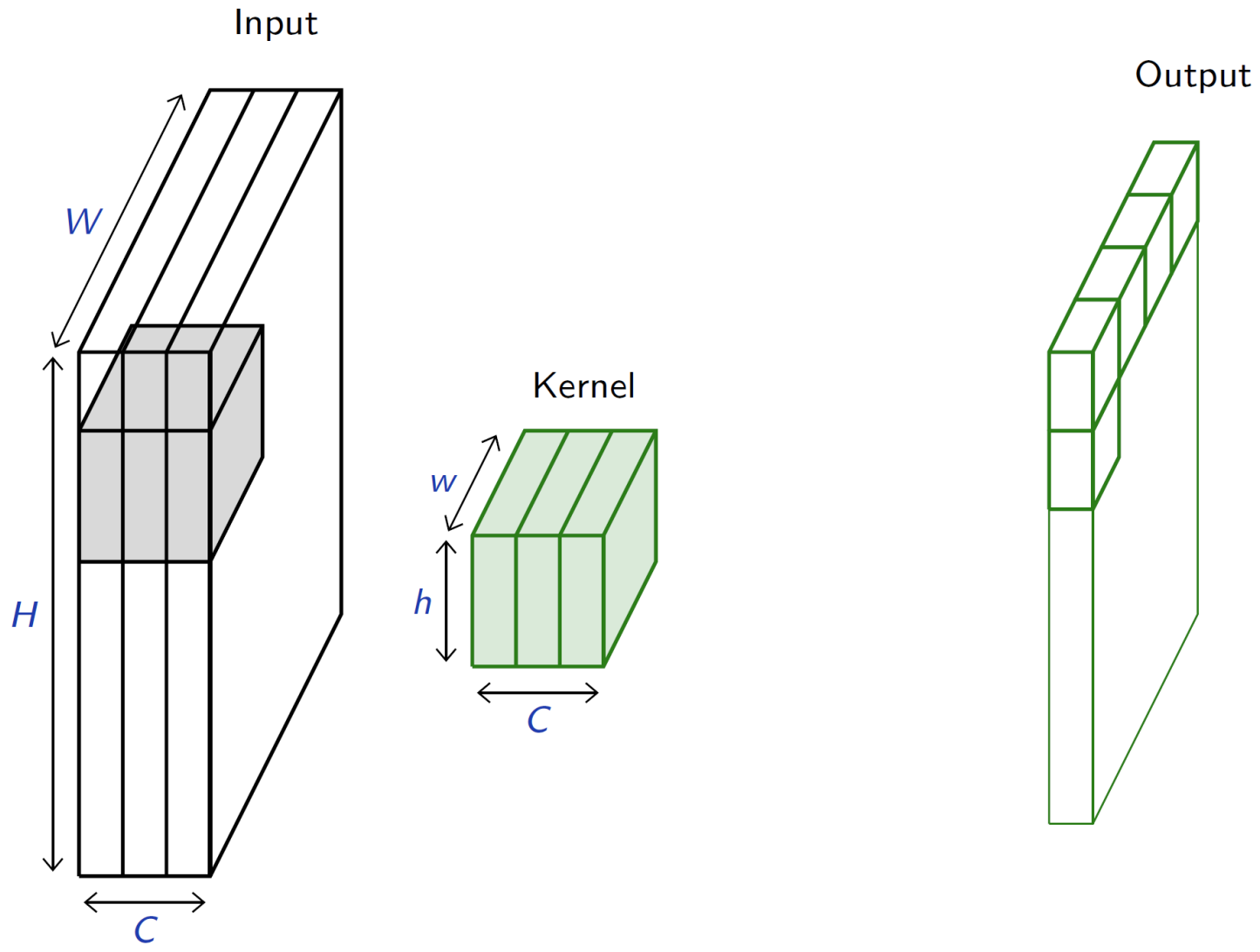
# Multi-channel 2D Convolution



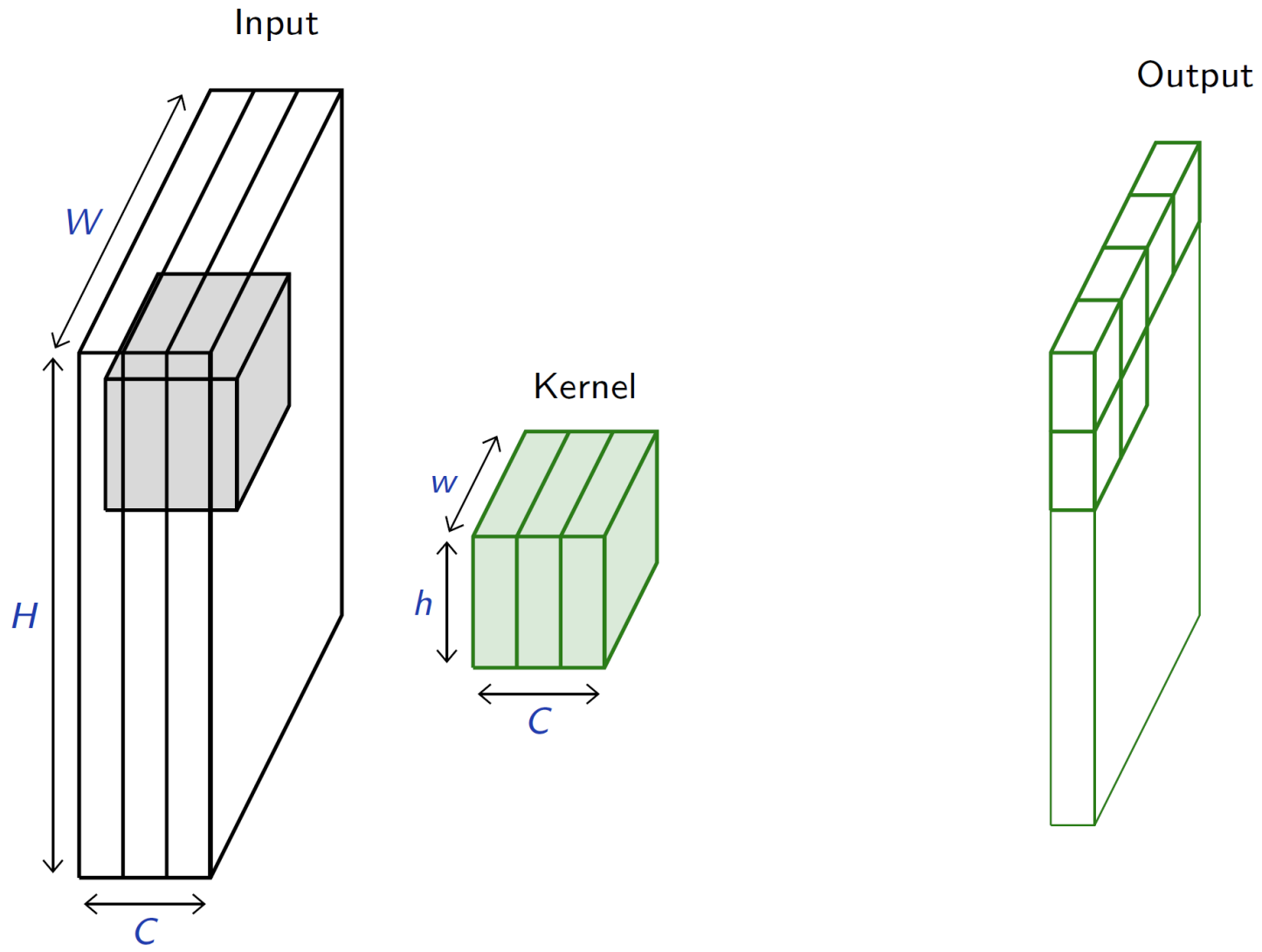
# Multi-channel 2D Convolution



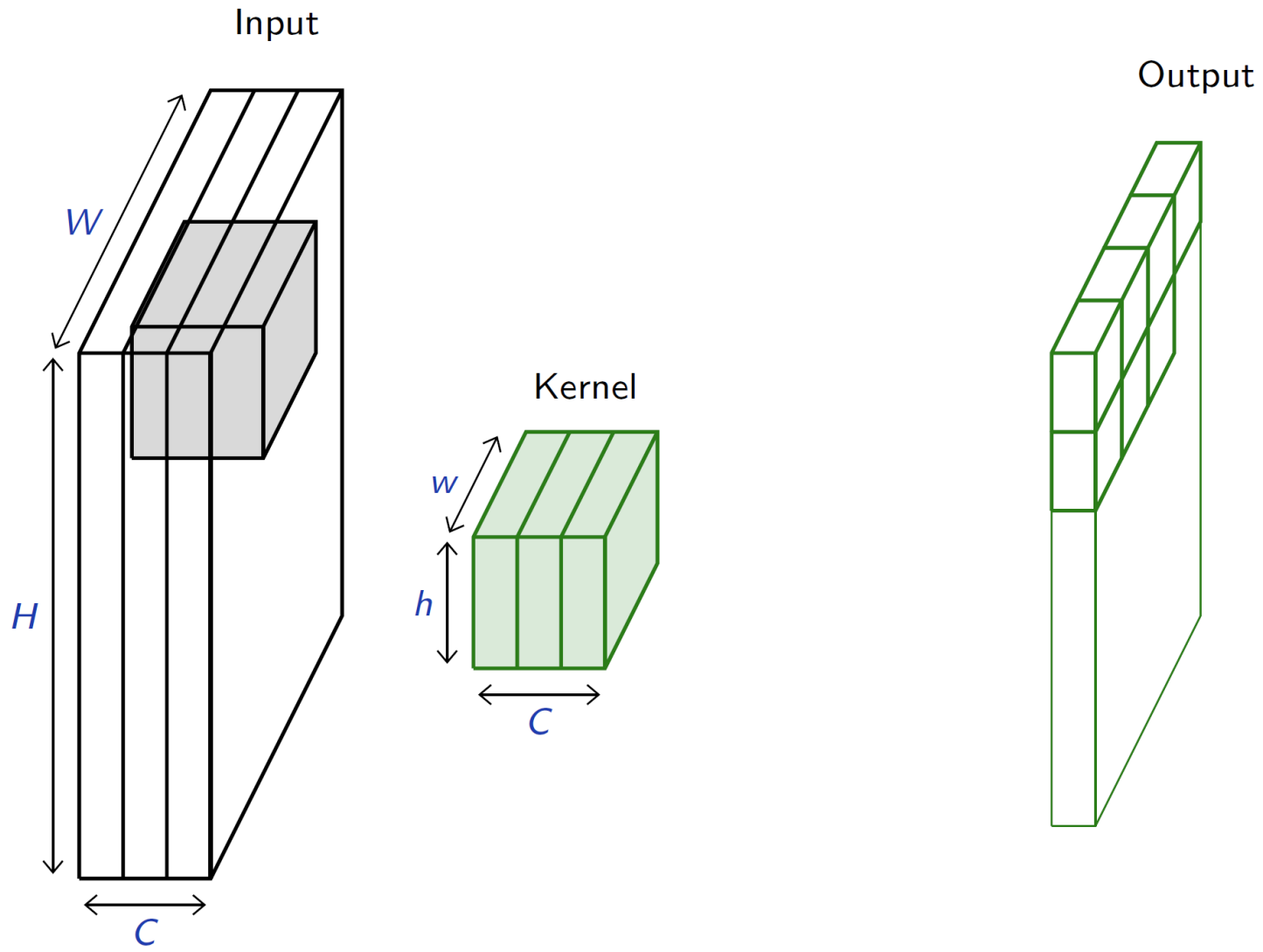
# Multi-channel 2D Convolution



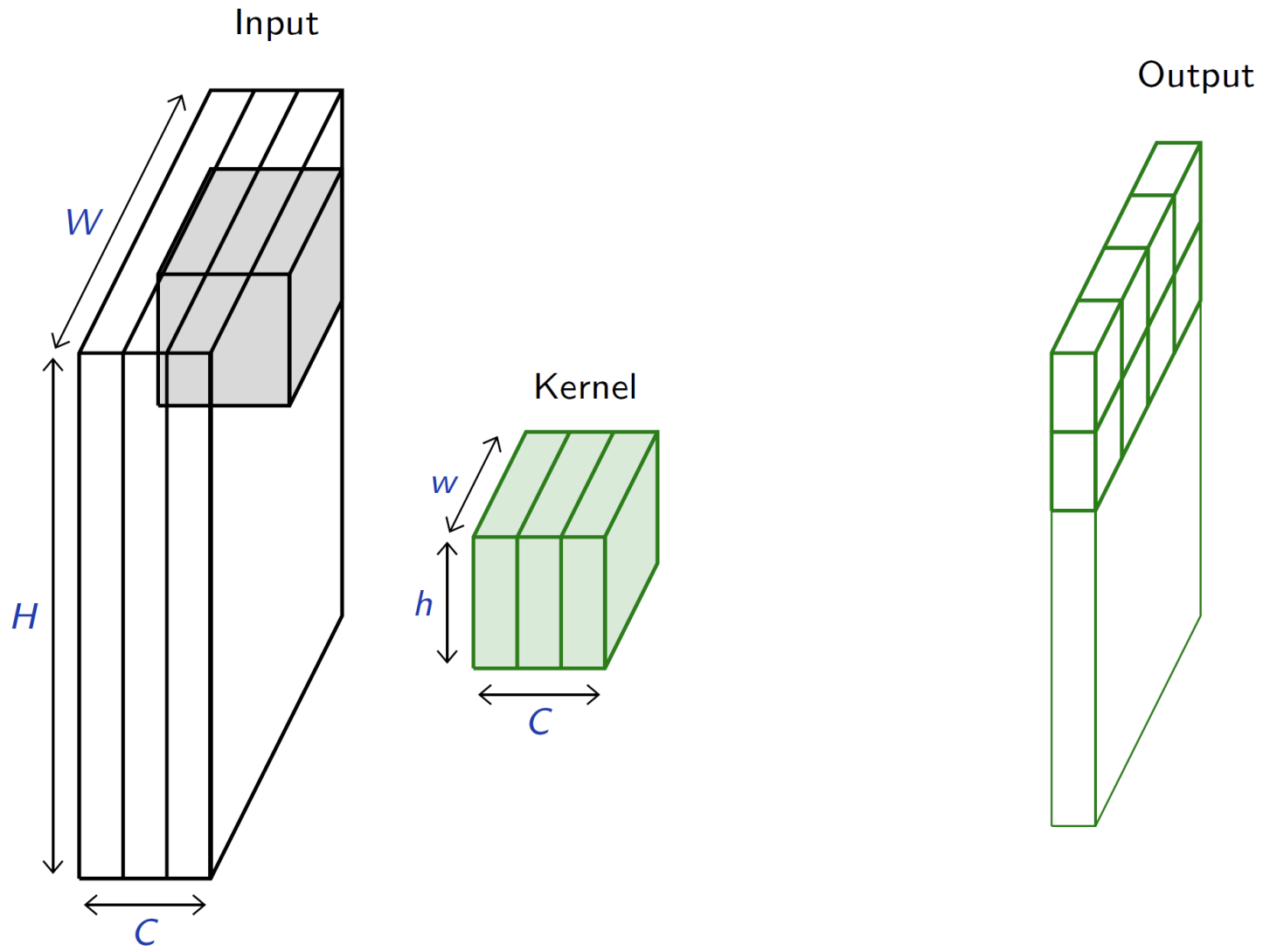
# Multi-channel 2D Convolution



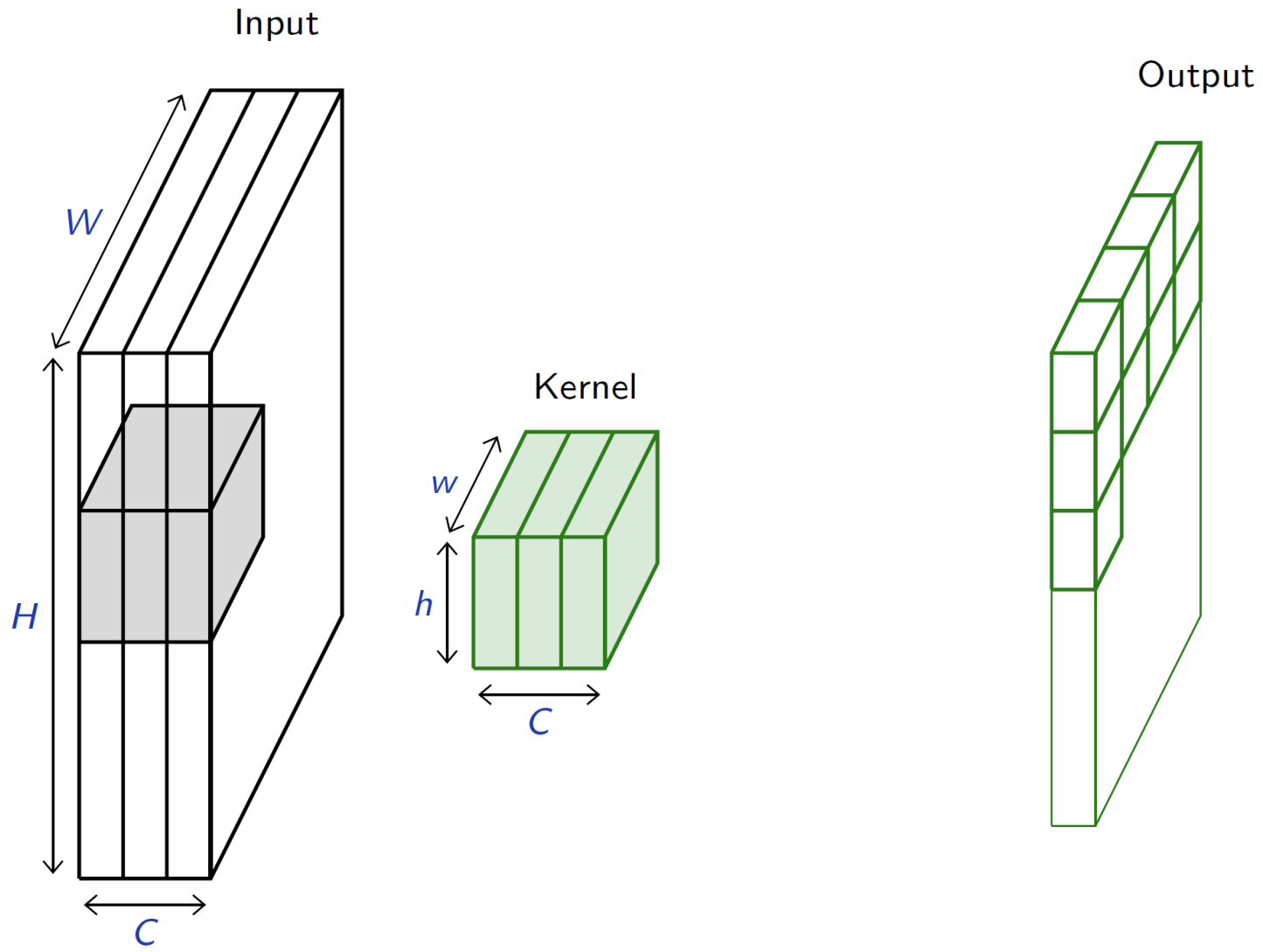
# Multi-channel 2D Convolution



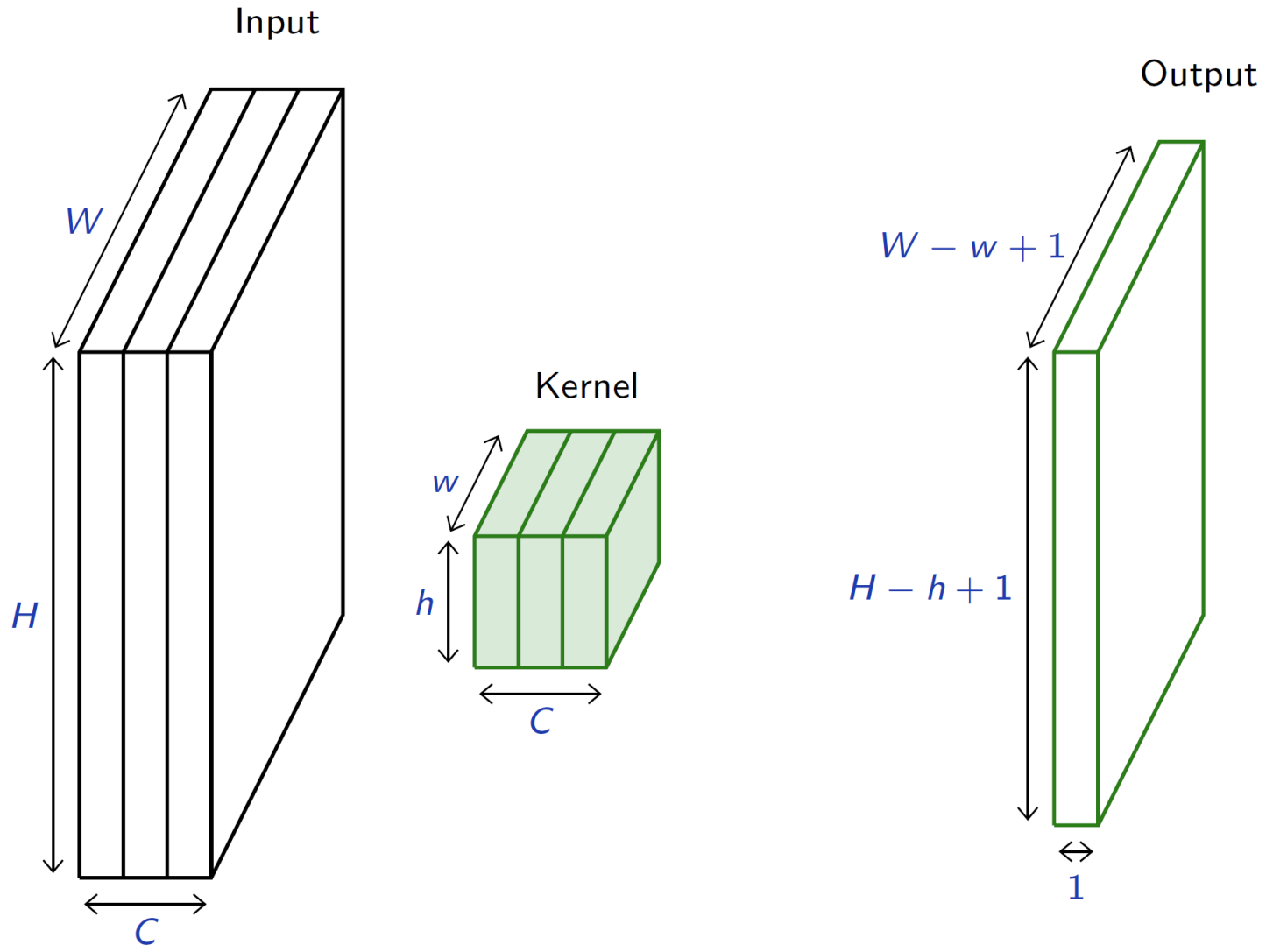
# Multi-channel 2D Convolution



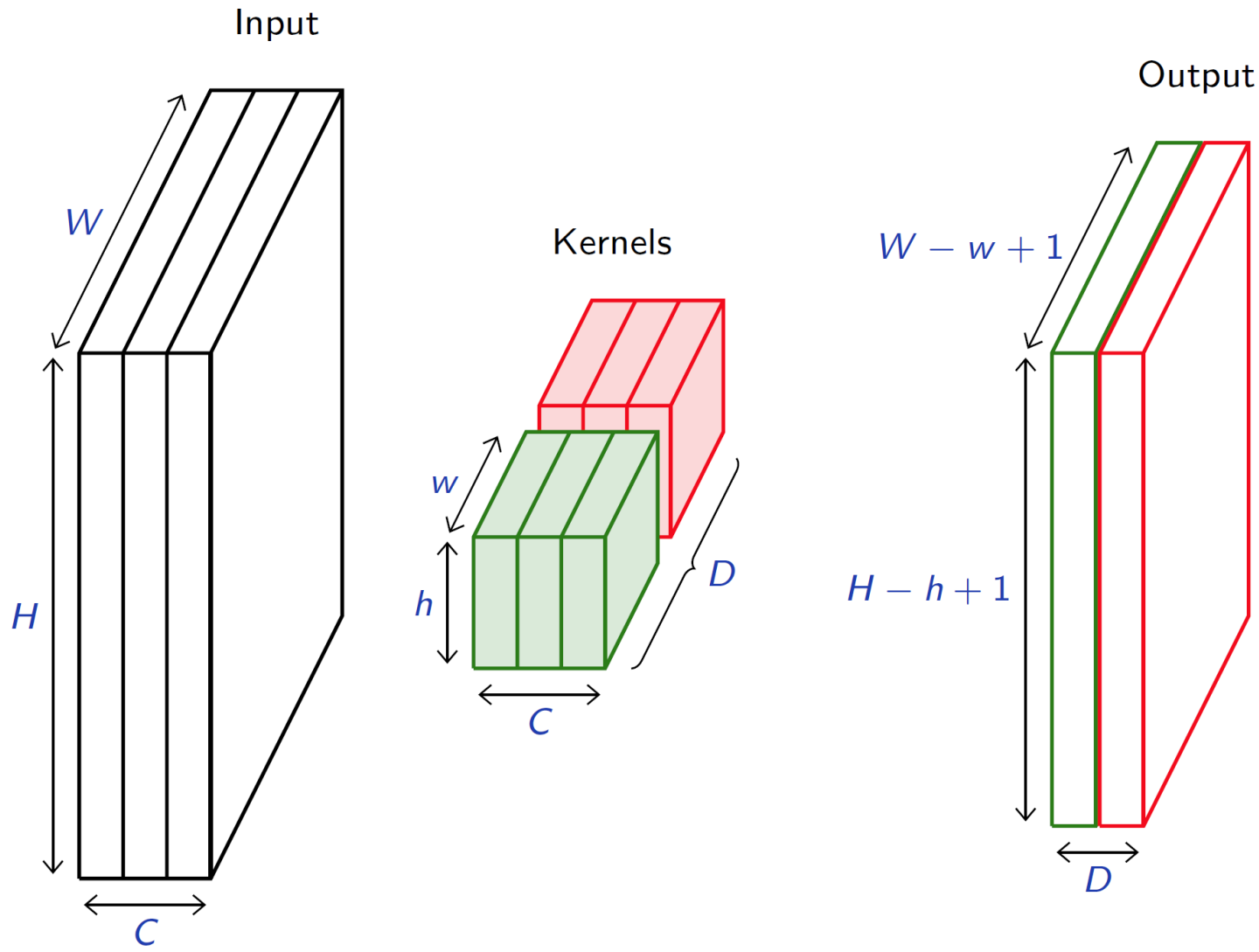
# Multi-channel 2D Convolution



# Multi-channel 2D Convolution



# Multi-channel and Multi-kernel 2D Convolution

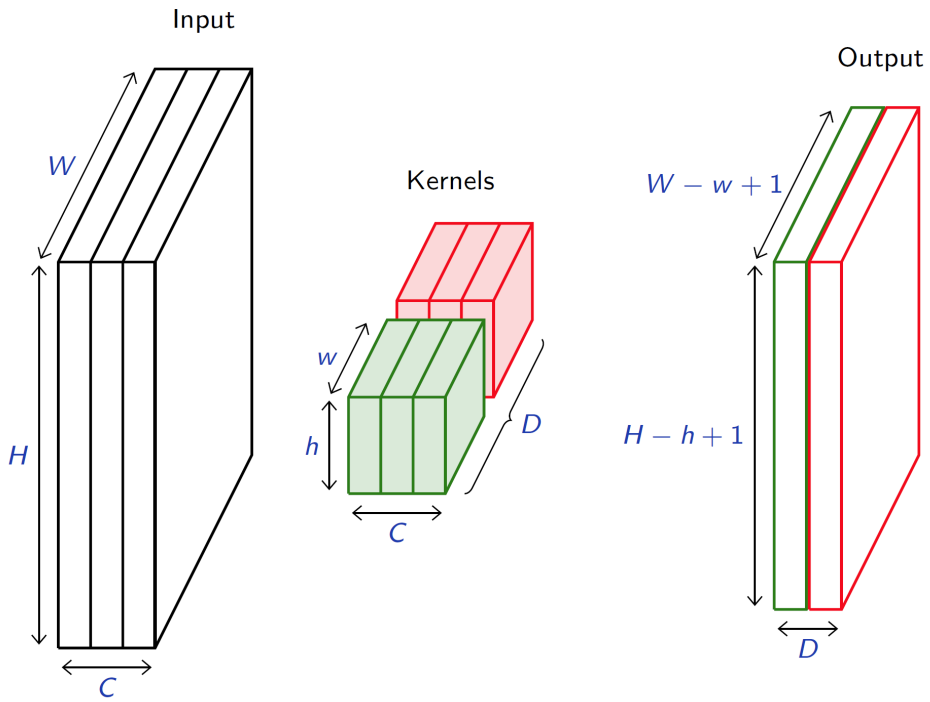


# Dealing with Shapes

- Activations or feature maps shape
  - Input $(W^i, H^i, C)$
  - Output $(W^o, H^o, D)$

- Kernel of Filter shape $(w, h, C, D)$
  - $w \times h$ Kernel size
  - $C$ Input channels
  - $D$ Output channels

- Numbers of parameters: $(w \times h \times C + 1) \times D$
  - bias



# 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.
- A 3D convolution can be used if the channel index has some metric meaning, such as time for a series of grayscale video frames. Otherwise swiping across channels makes no sense.

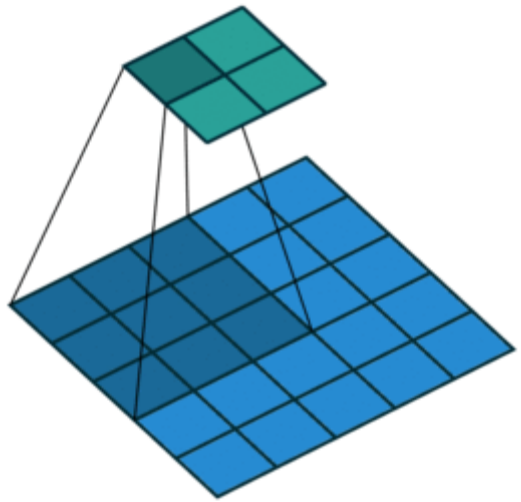
- We usually refer to one of the channels generated by a convolution layer as an activation map.
- The sub-area of an input map that influences a component of the output as the receptive field of the latter.
- In the context of convolutional networks, a standard linear layer is called a fully connected layer since every input influences every output.

# Padding and Stride

- Convolutions have two additional standard parameters:
  - The padding specifies the size of a zeroed frame added around the input
  - the stride specifies a step size when moving the kernel across the signal.

# Strides

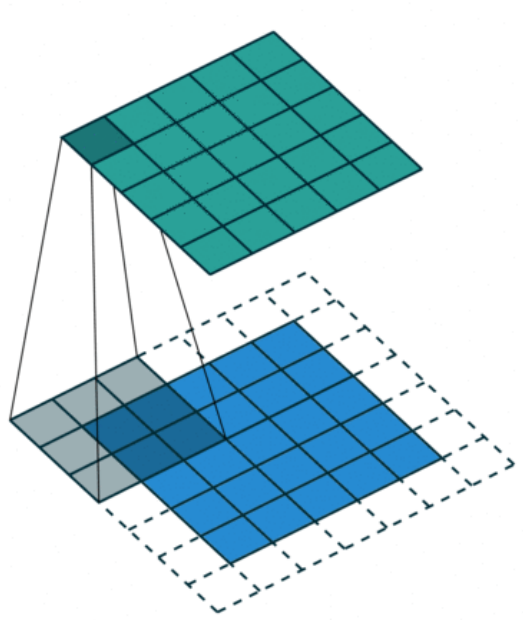
- Strides: increment step size for the convolution operator
- Reduces the size of the output map



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

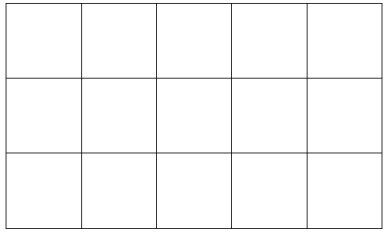
# Padding

- 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



# Padding and Stride

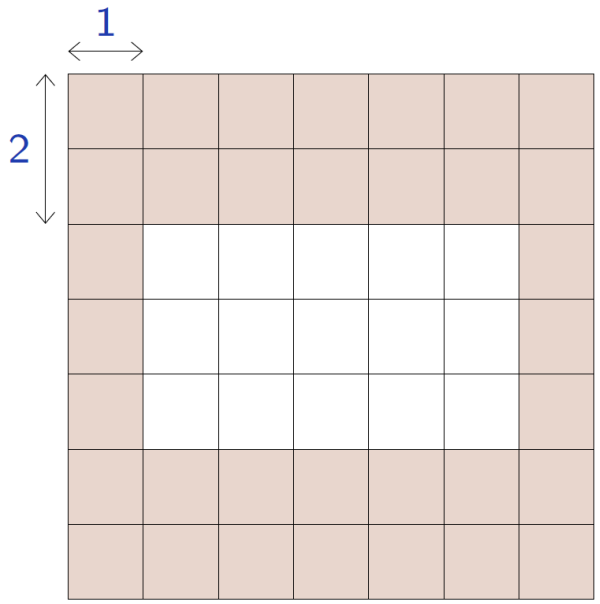
- Here with $3 \times 5 \times C$ as input



Input

# Padding and Stride

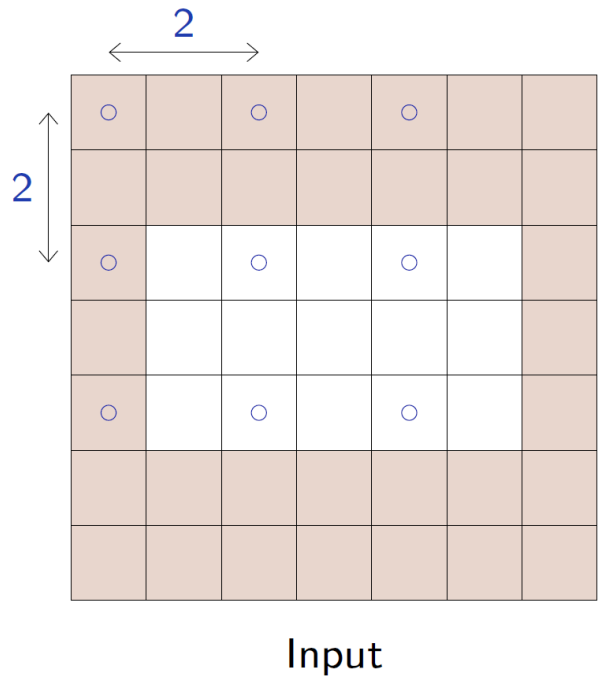
- Here with $3 \times 5 \times C$ as input, a padding of $(2,1)$




Input

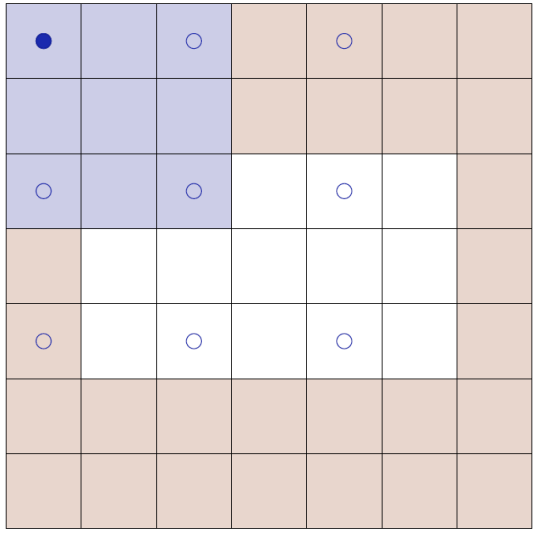
# Padding and Stride

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

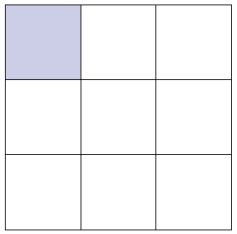


# Padding and Stride

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



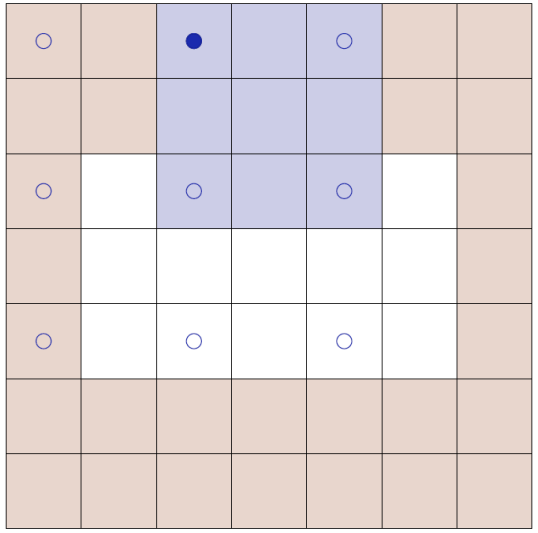
Input



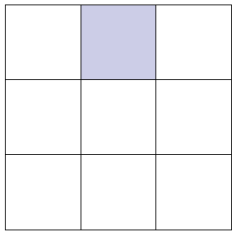
Output

# Padding and Stride

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



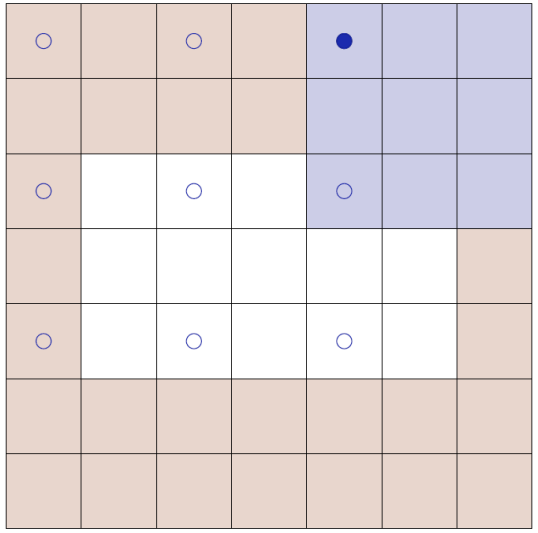
Input



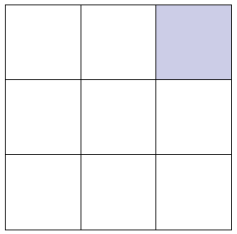
Output

# Padding and Stride

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



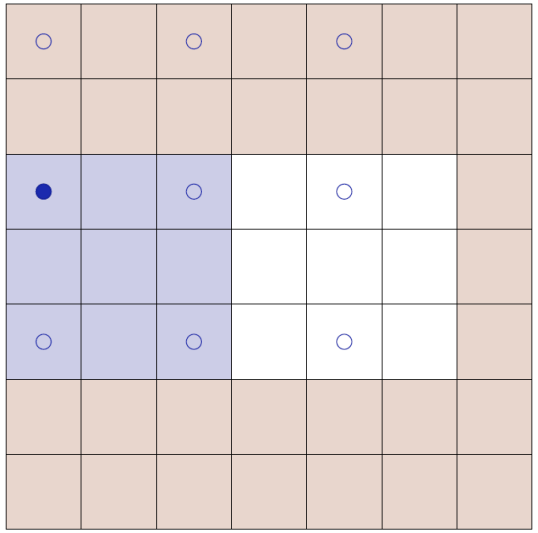
Input



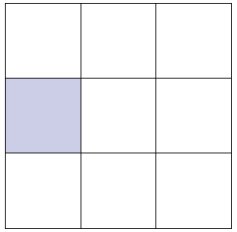
Output

# Padding and Stride

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



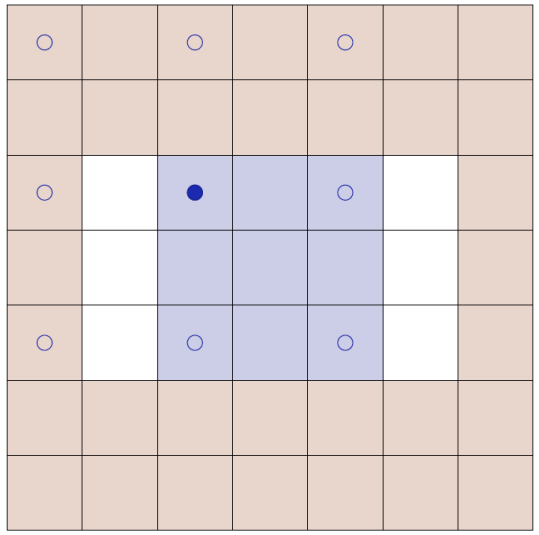
Input



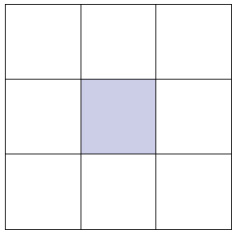
Output

# Padding and Stride

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



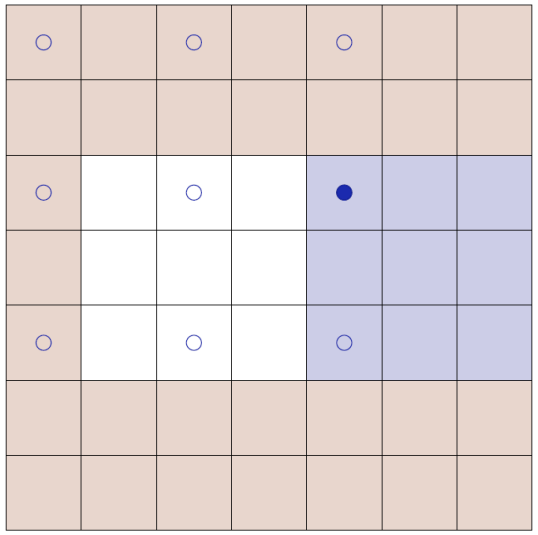
Input



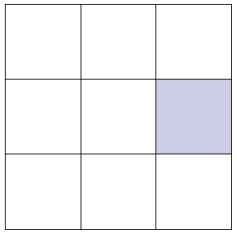
Output

# Padding and Stride

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



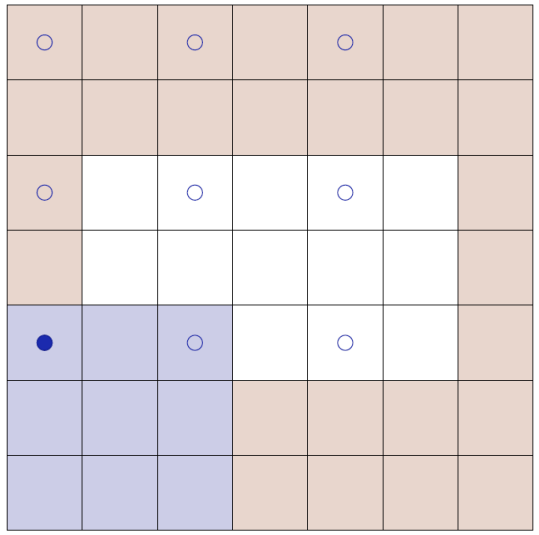
Input



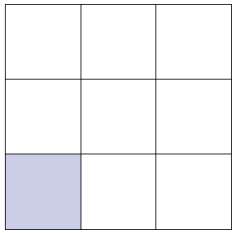
Output

# Padding and Stride

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



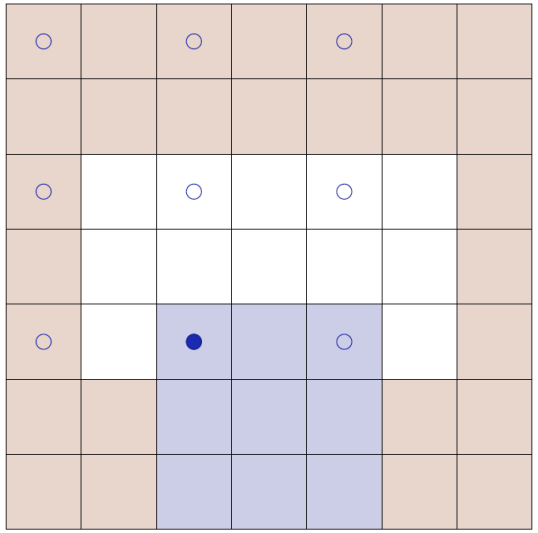
Input



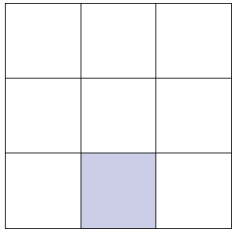
Output

# Padding and Stride

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



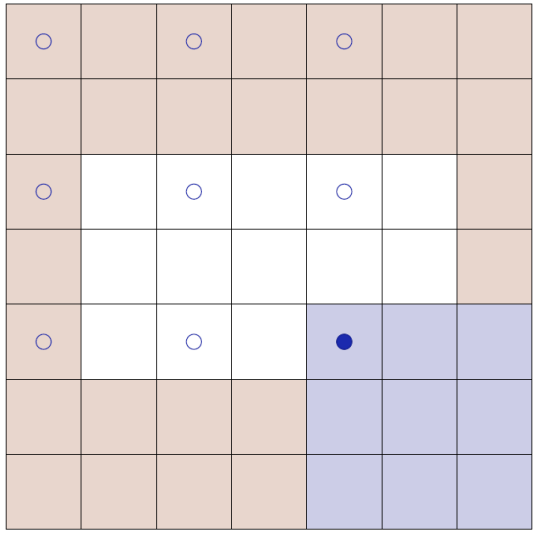
Input



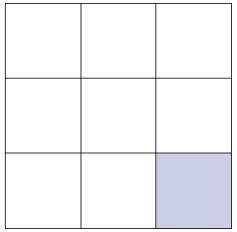
Output

# Padding and Stride

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

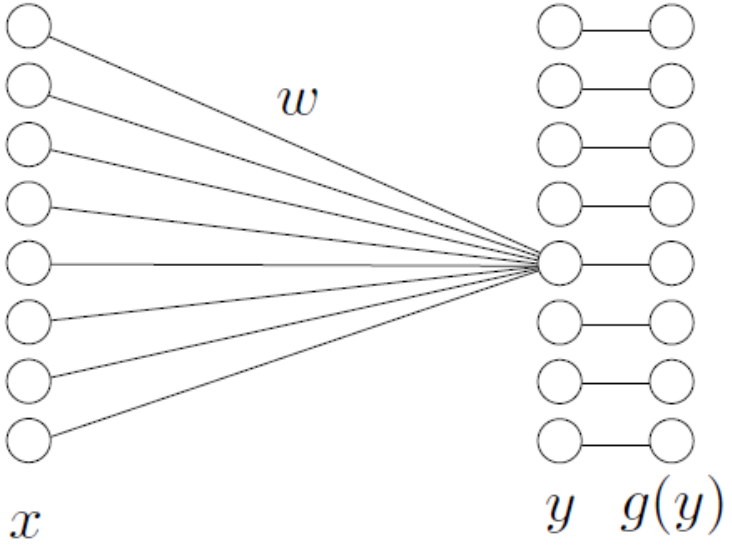


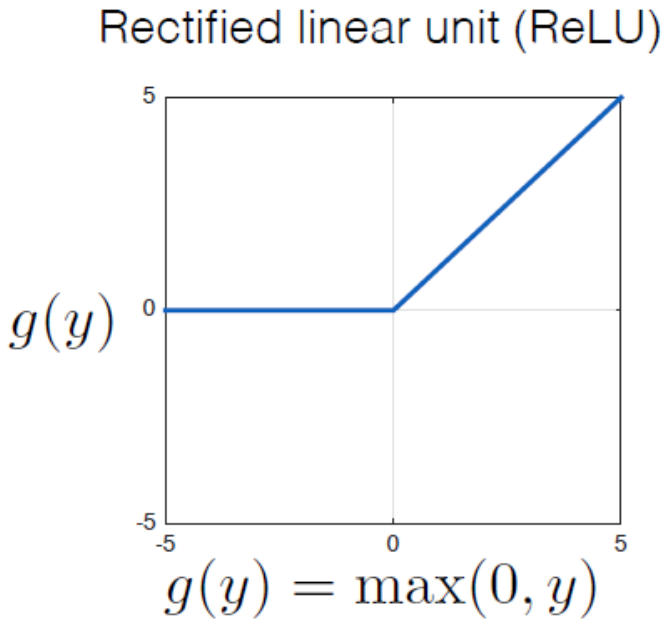
Input



Output

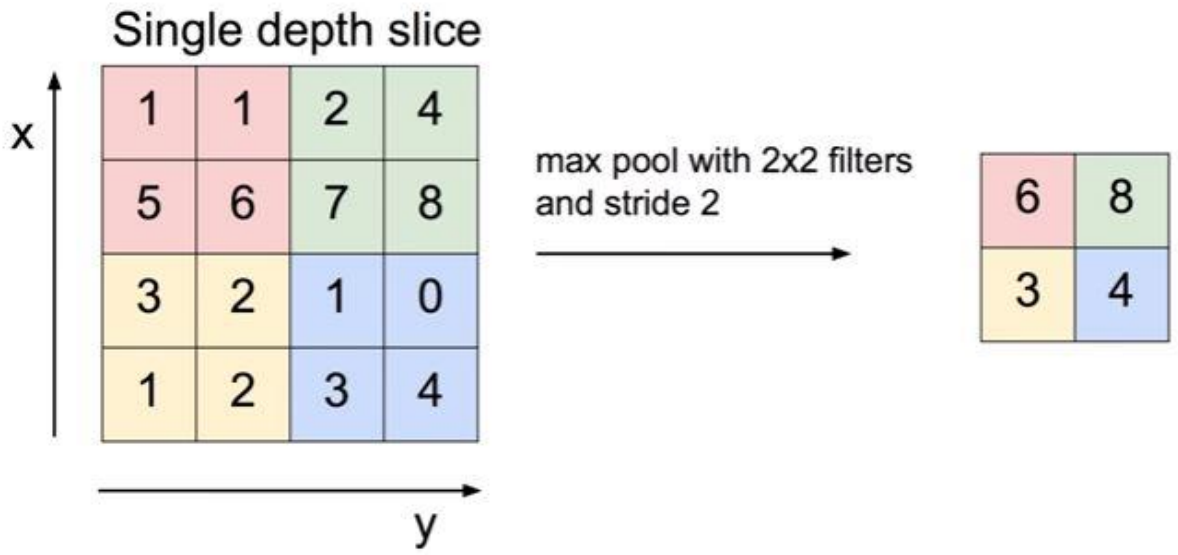
# Nonlinear Activation Function



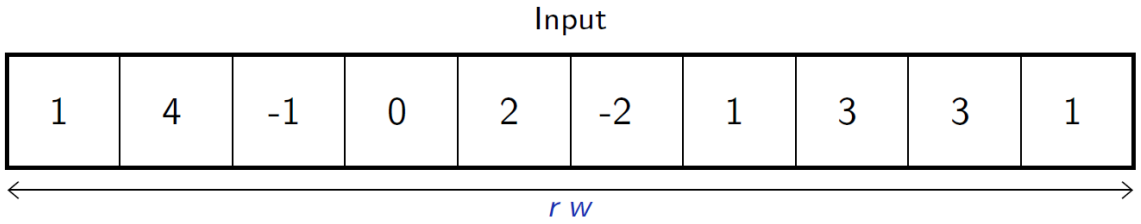
# Pooling

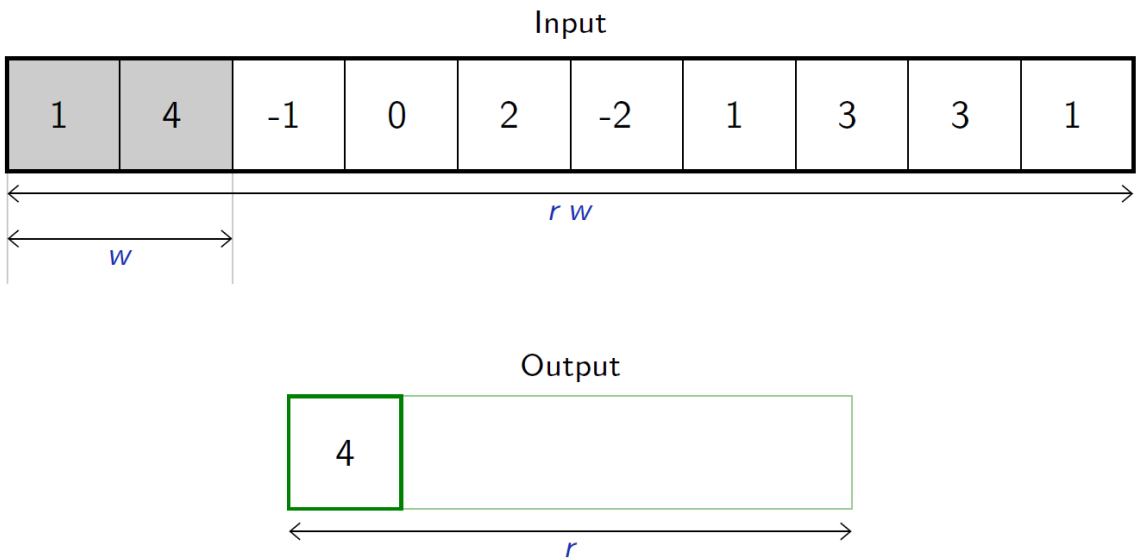
- Compute a maximum value in a sliding window (max pooling)
- Reduce spatial resolution for faster computation
- Achieve invariance to local translation
- Max pooling introduces invariances
  - Pooling size : 2×2
  - No parameters: max or average of 2x2 units

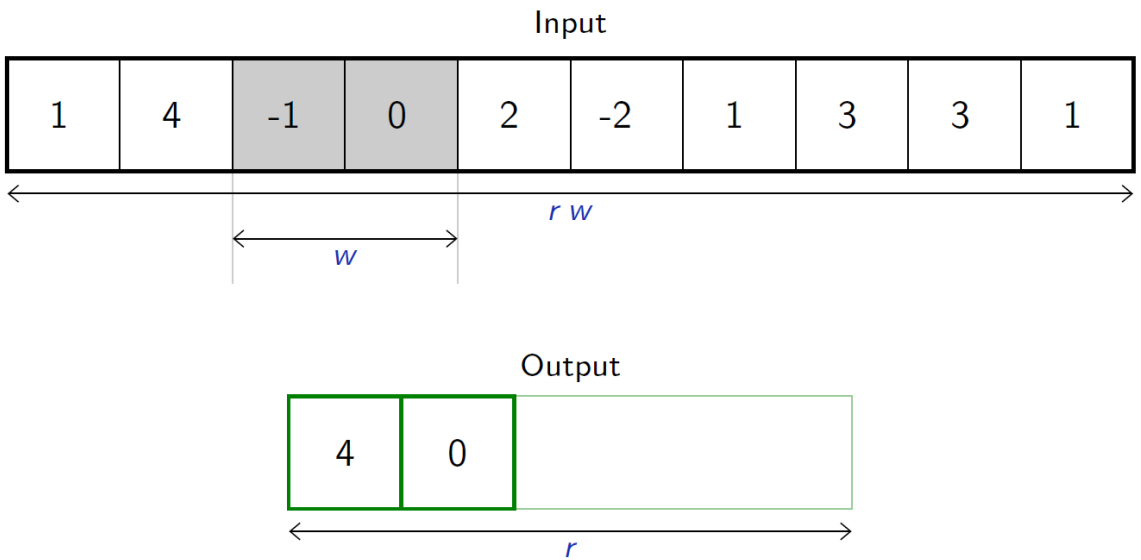


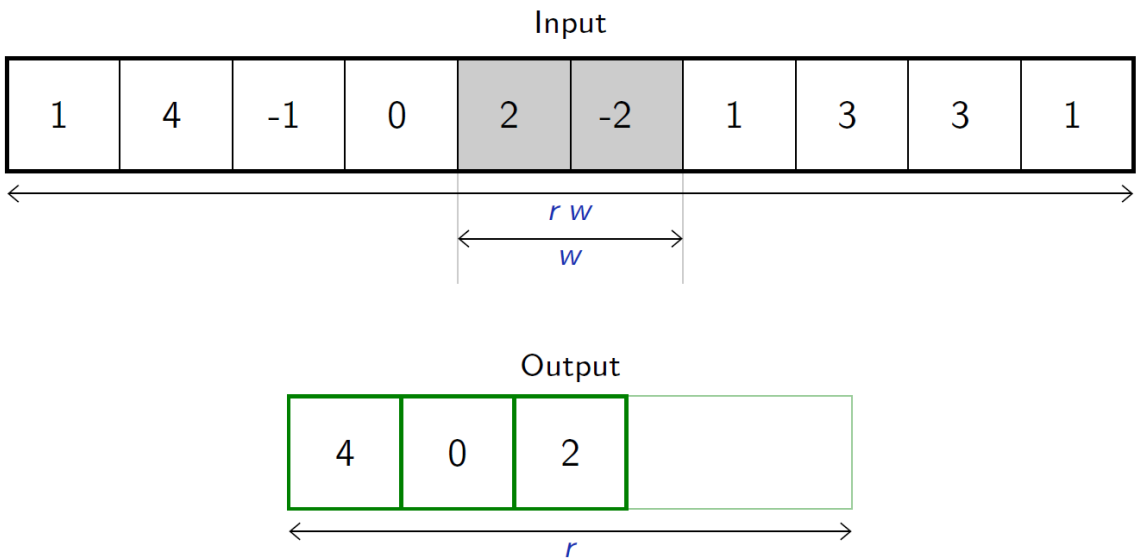
# Pooling

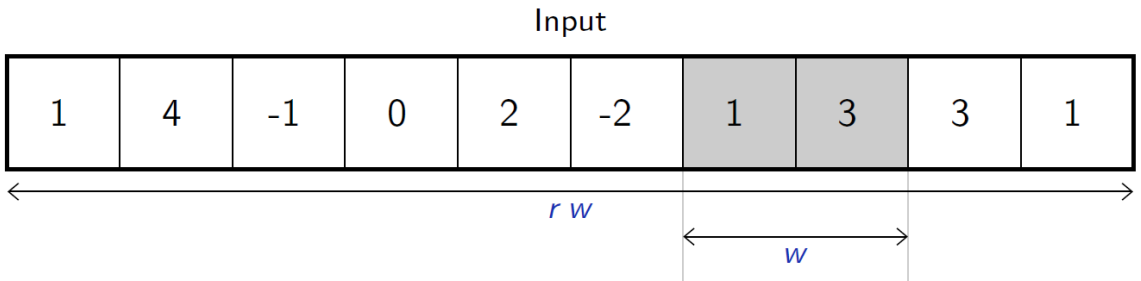
- The most standard type of pooling is the max-pooling, which computes max values over non-overlapping blocks
- For instance in 1d with a kernel of size 2

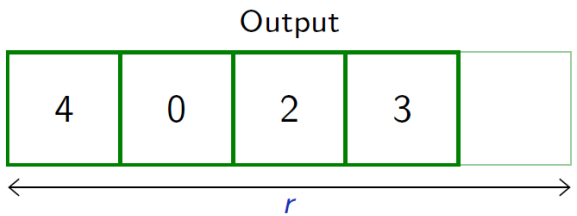


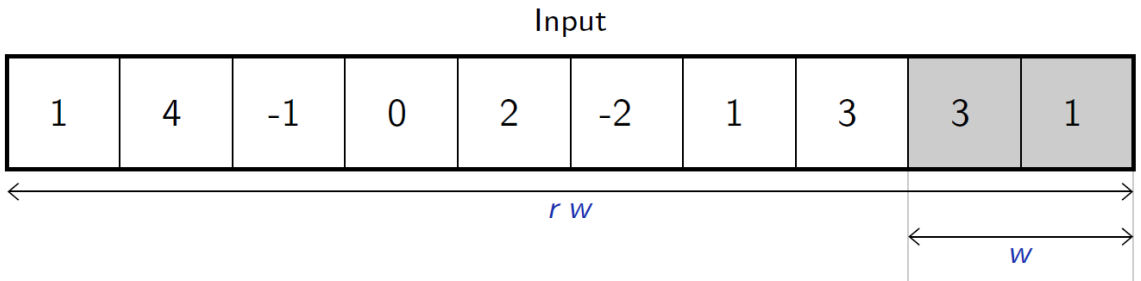

Input
1
4
-1
0
2
-2
1
3
3
1
r w
w
Output
4
r

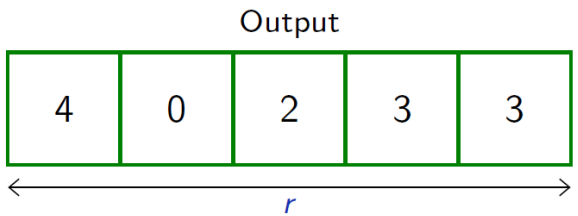

Input
1
4
-1
0
2
-2
1
3
3
1
r w
w
Output
4
0
r


Input
1
4
-1
0
2
-2
1
3
3
1
r w
w
Output
4
0
2
r

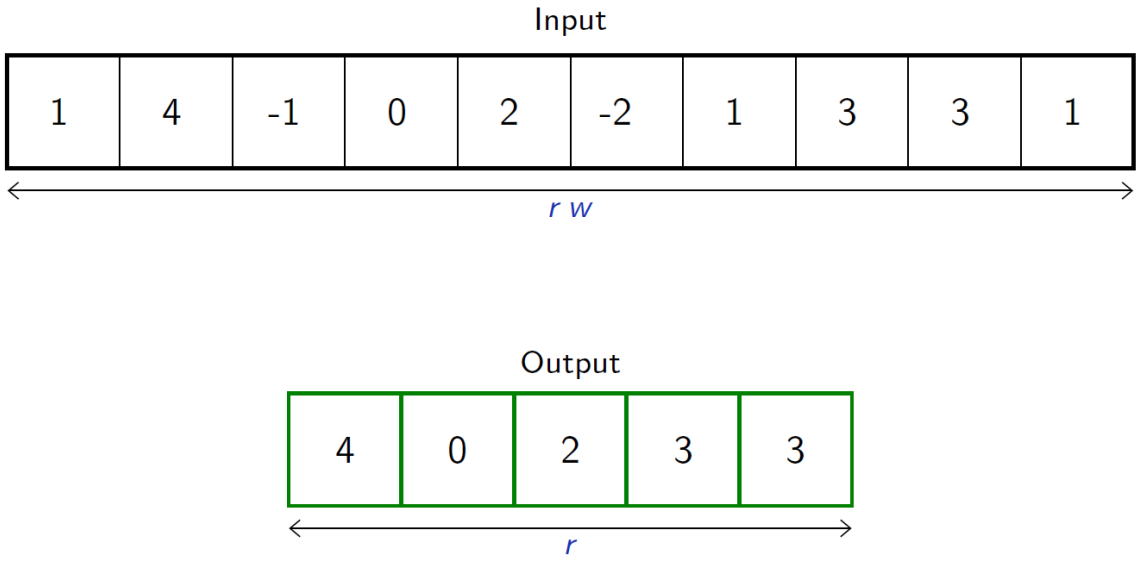

Input
1
4
-1
0
2
-2
1
3
3
1
r w
w


Output
4
0
2
3
r


Input
1
4
-1
0
2
-2
1
3
3
1
r w
w


Output
4
0
2
3
3
r

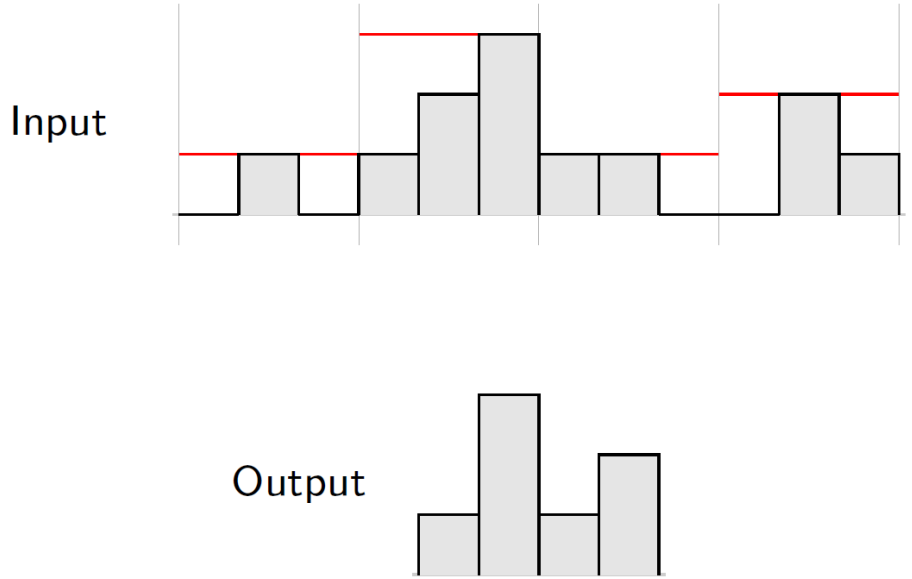
- Such an operation aims at grouping several activations into a single “more meaningful” one.




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

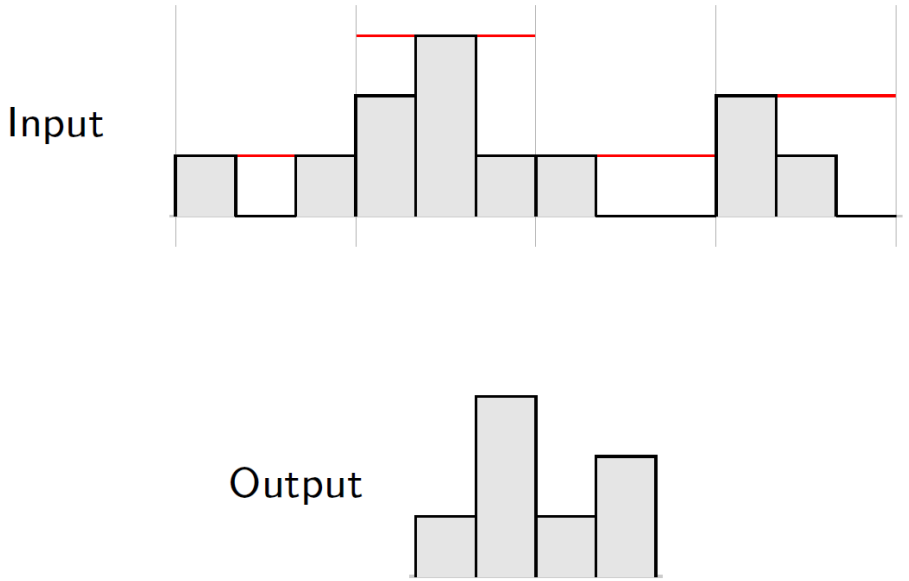
# Pooling: Invariance

- 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

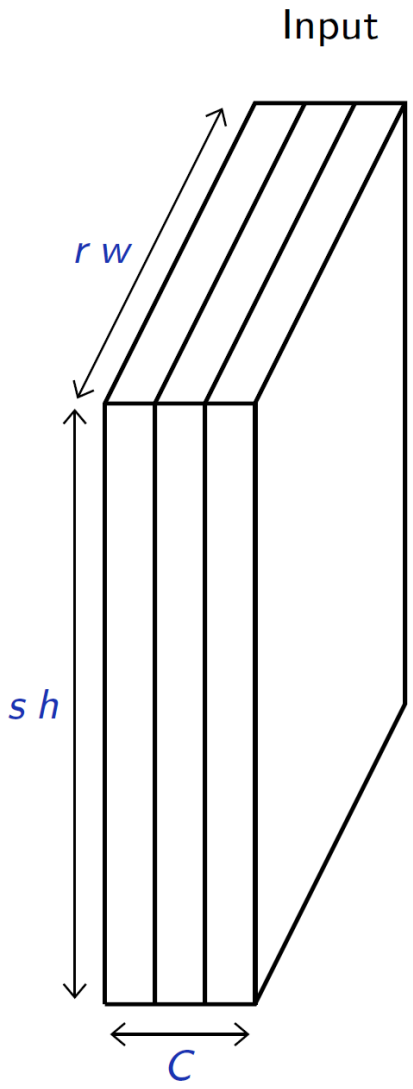


# Pooling: Invariance

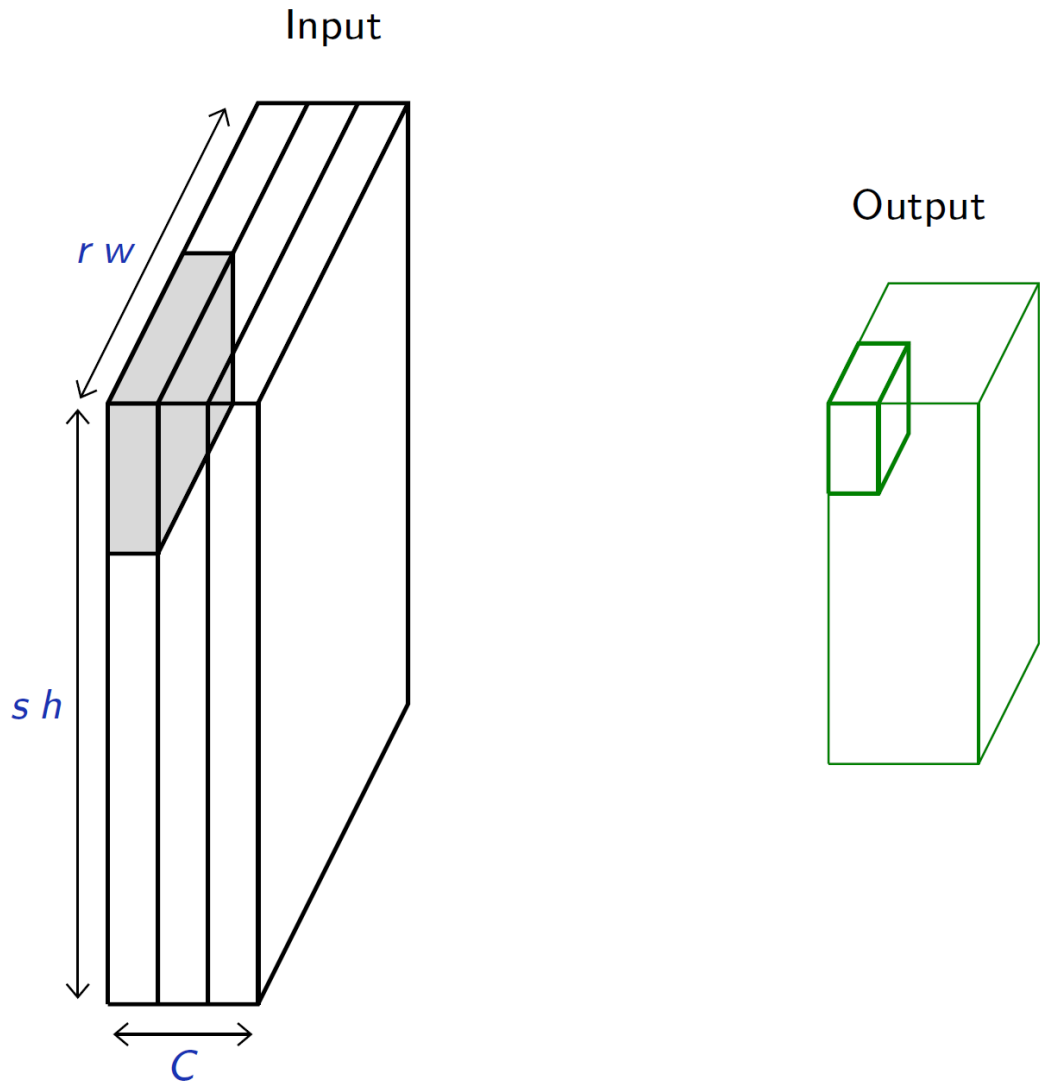
- Pooling provides invariance to any permutation inside one of the cell
- More practically, it provides a pseudo-invariance to deformations that result into local translations



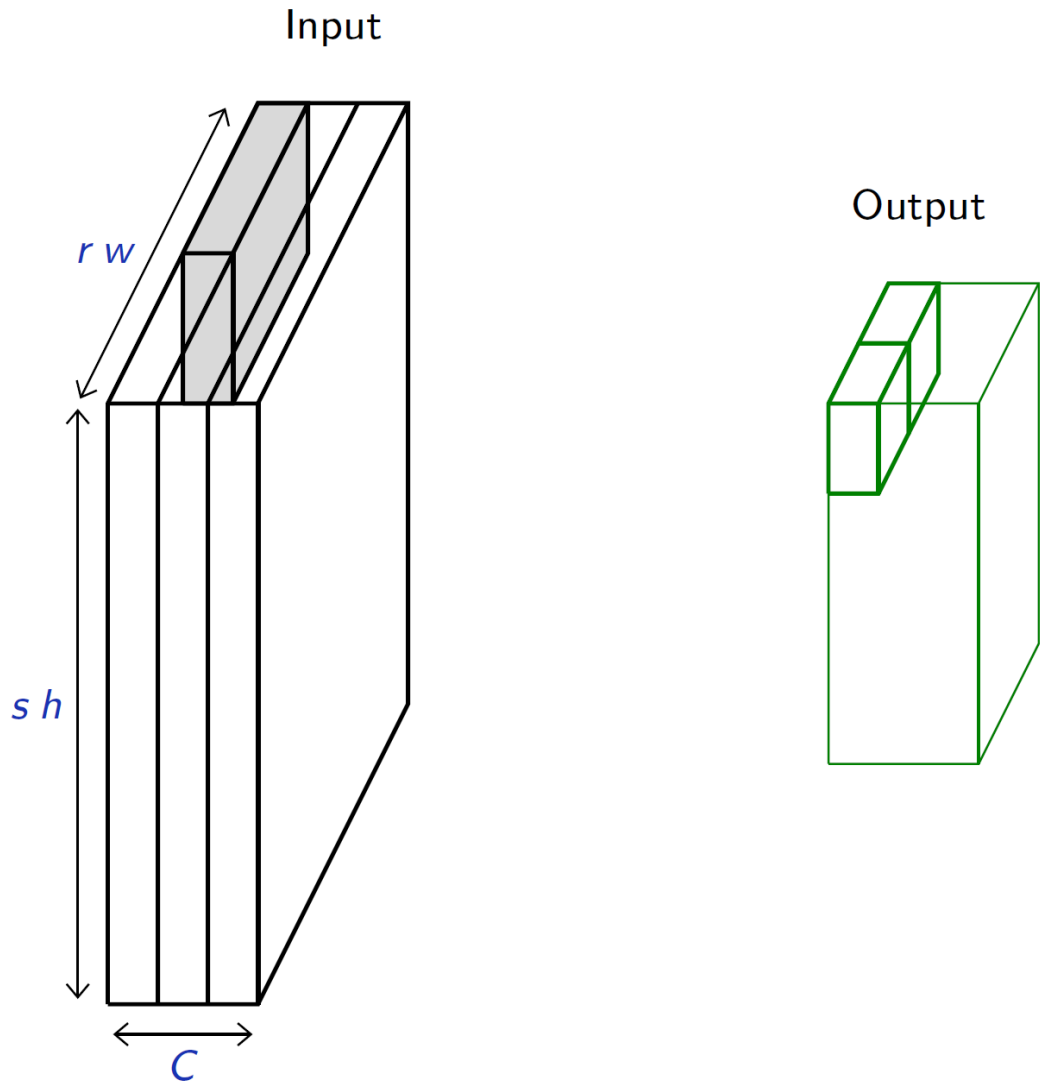
# Multi-channel Pooling



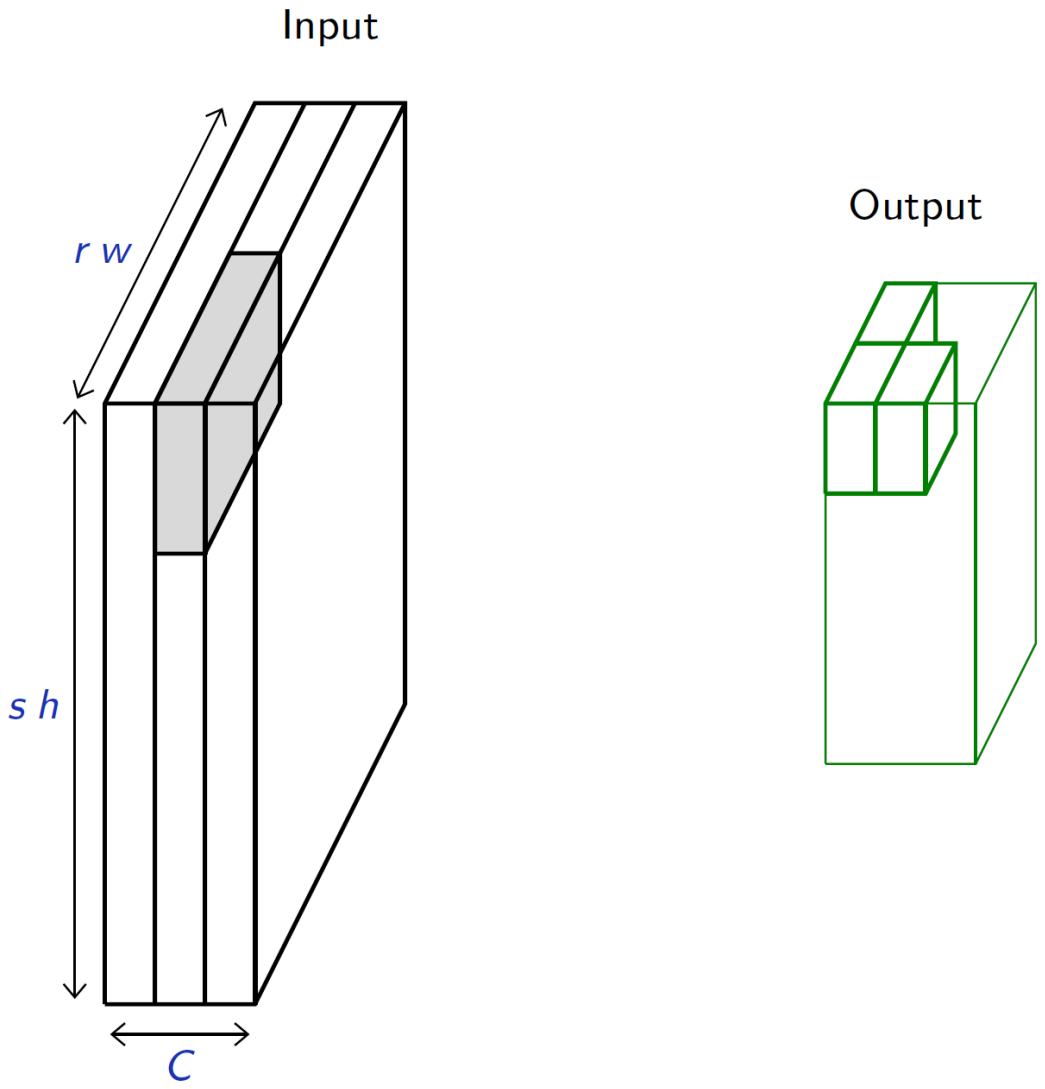
# Multi-channel Pooling



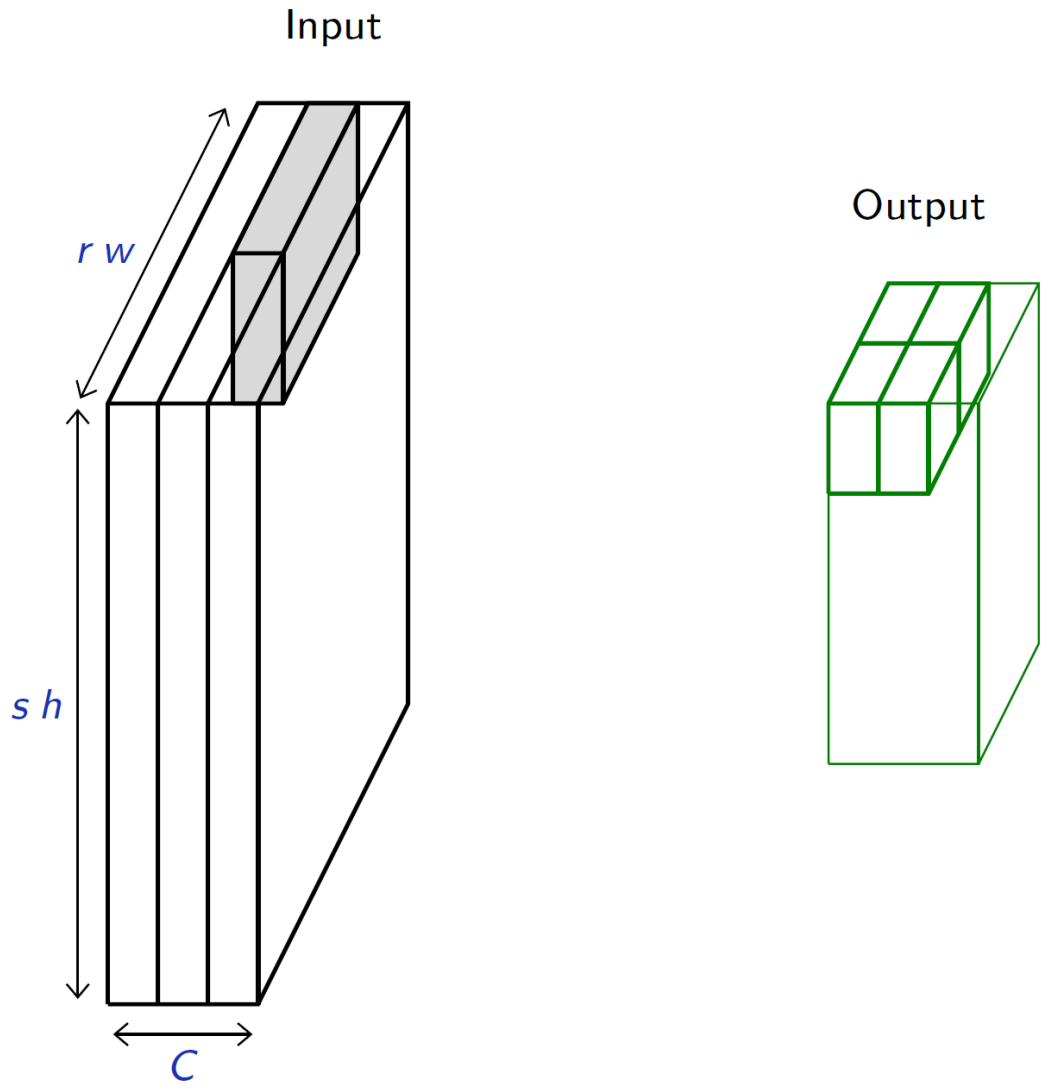
# Multi-channel Pooling



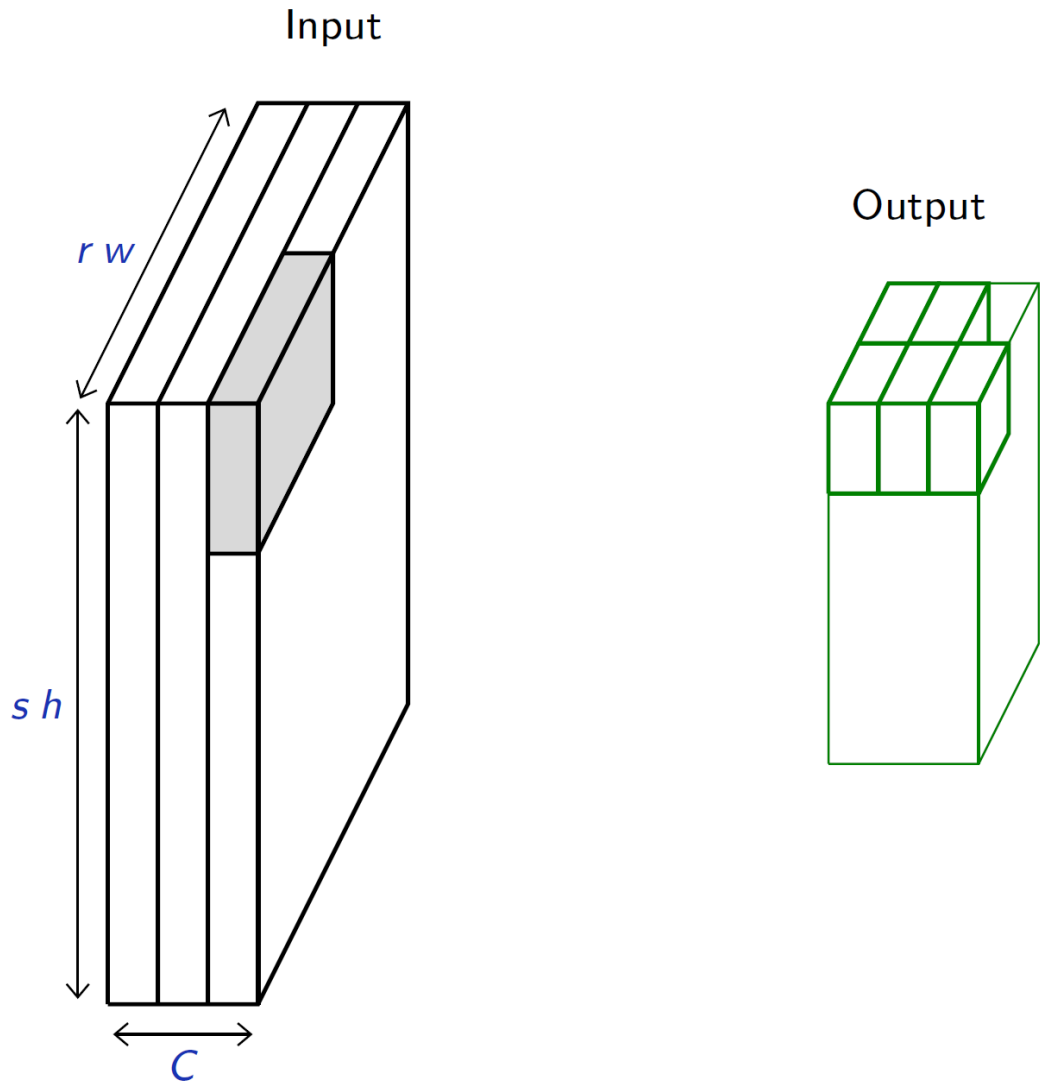
# Multi-channel Pooling



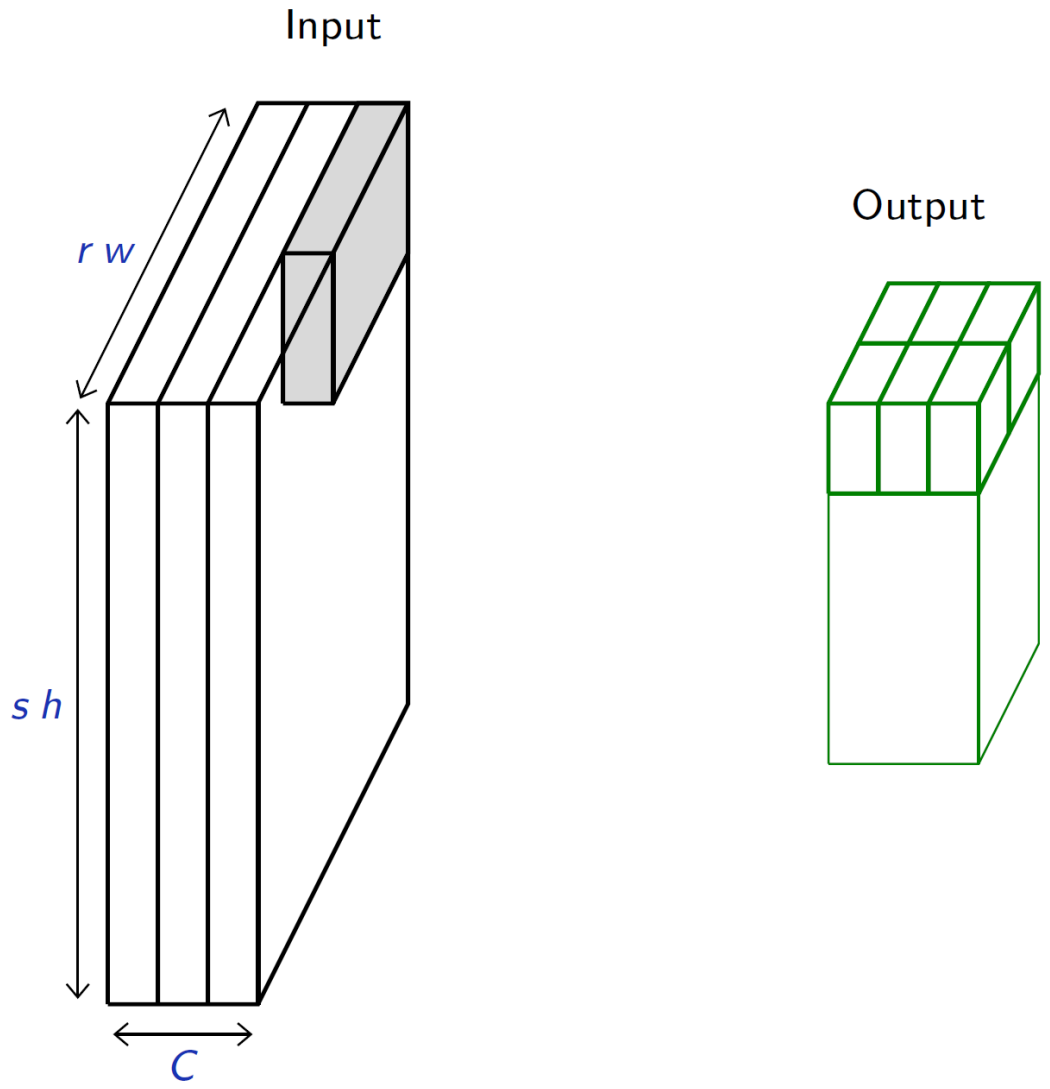
# Multi-channel Pooling



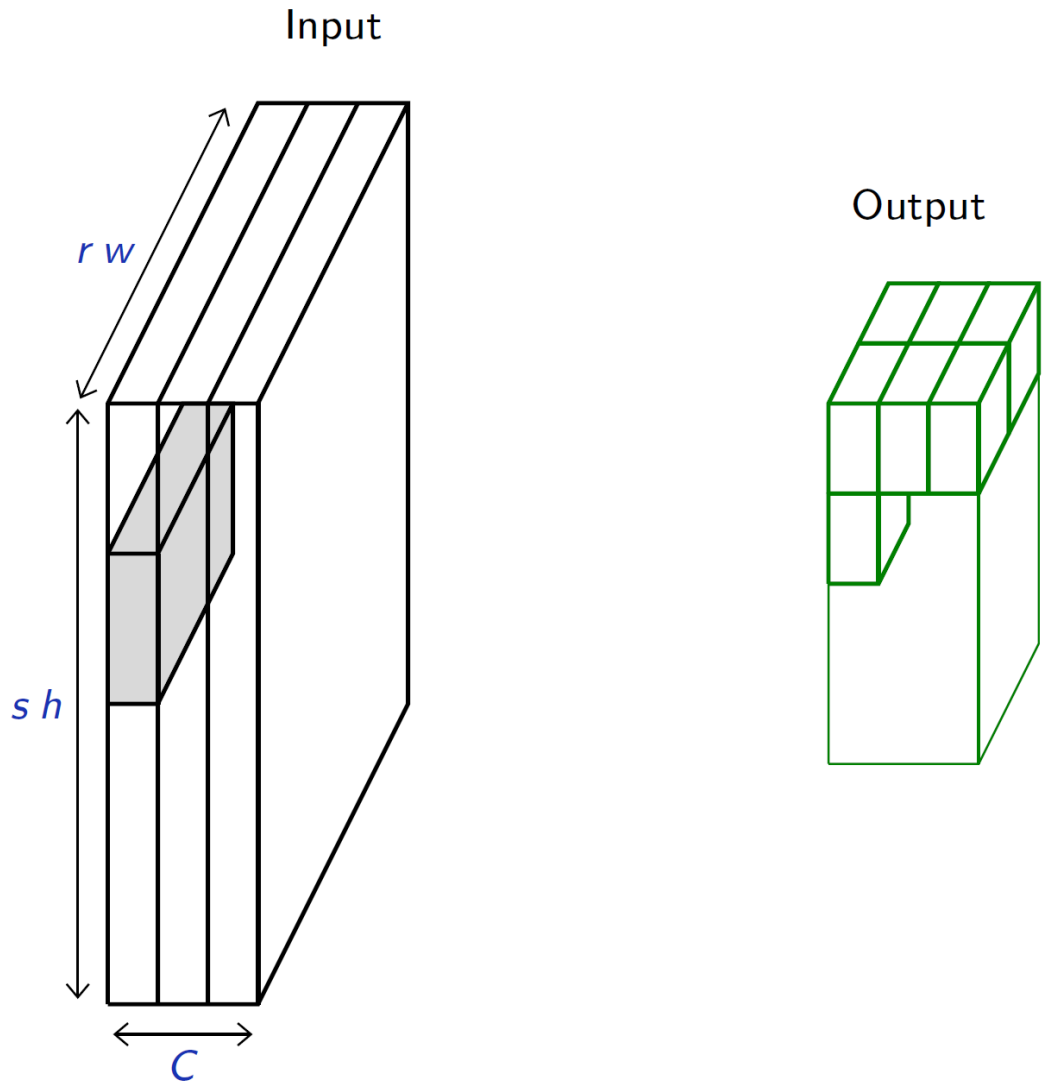
# Multi-channel Pooling



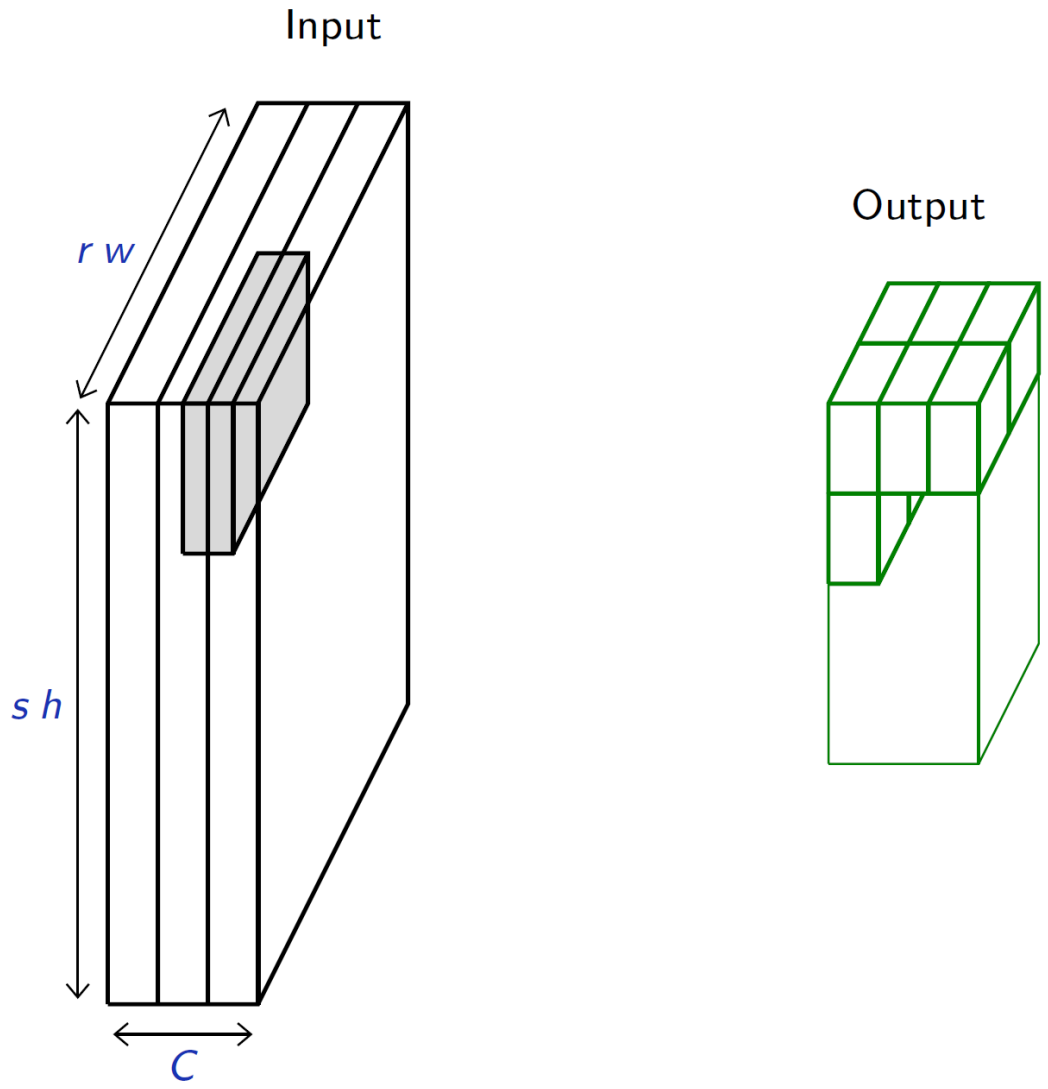
# Multi-channel Pooling



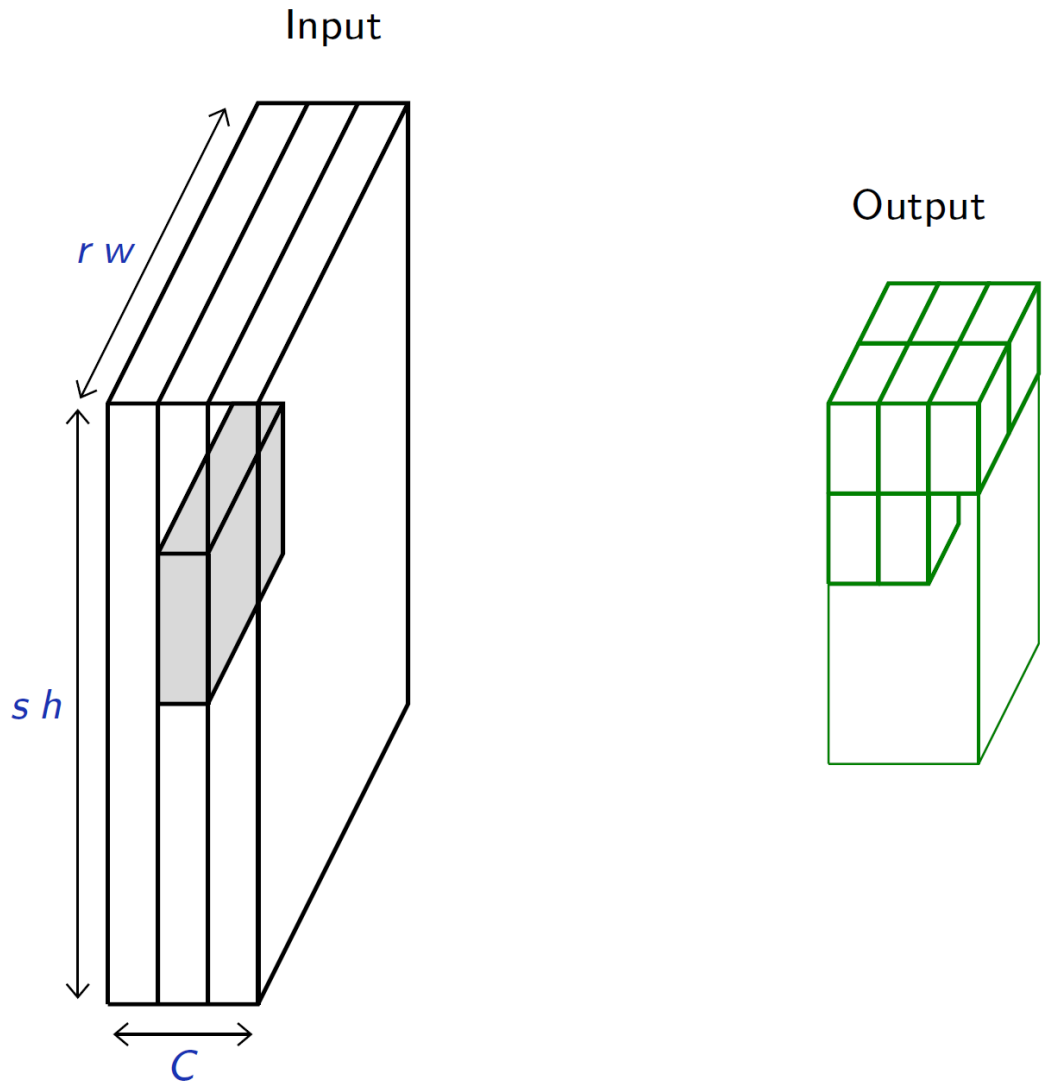
# Multi-channel Pooling



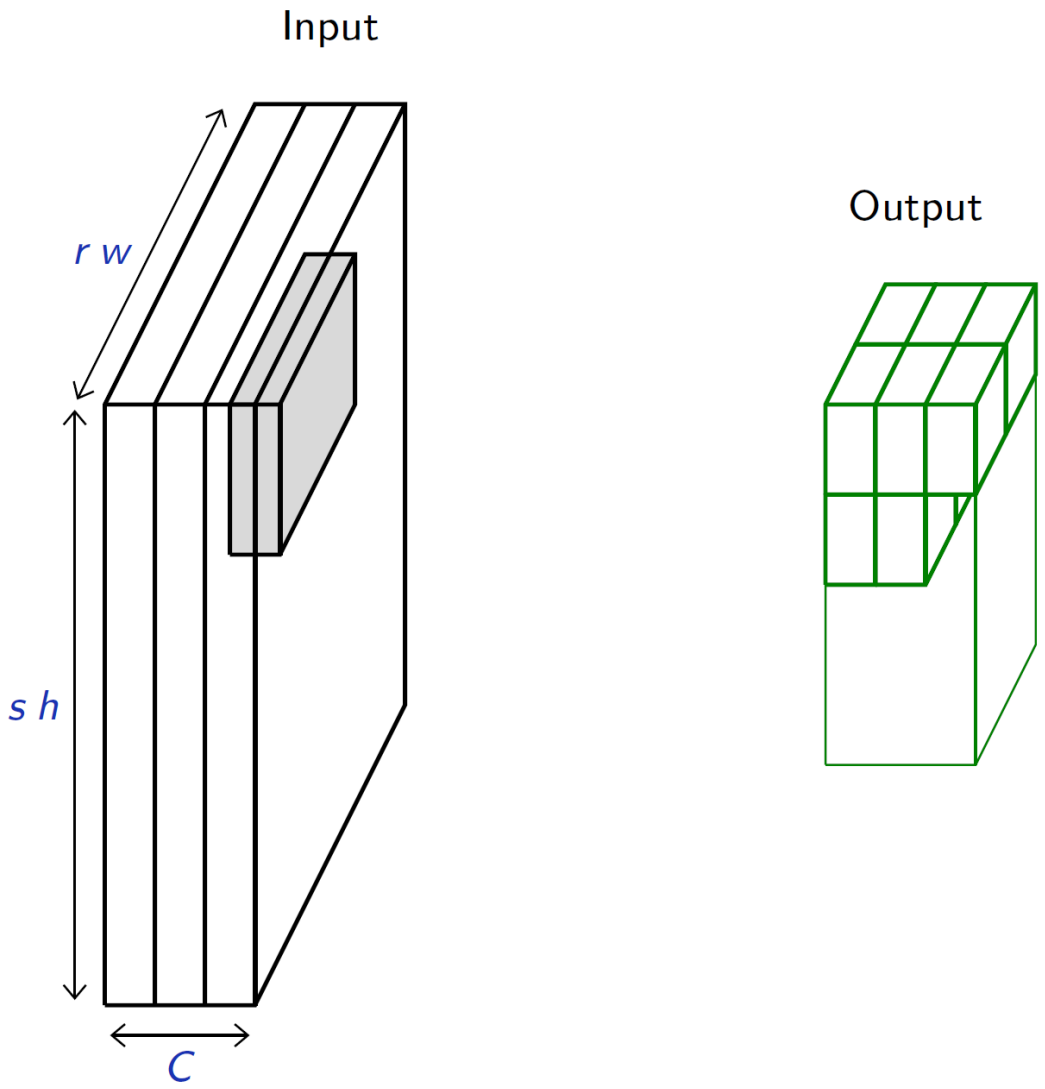
# Multi-channel Pooling



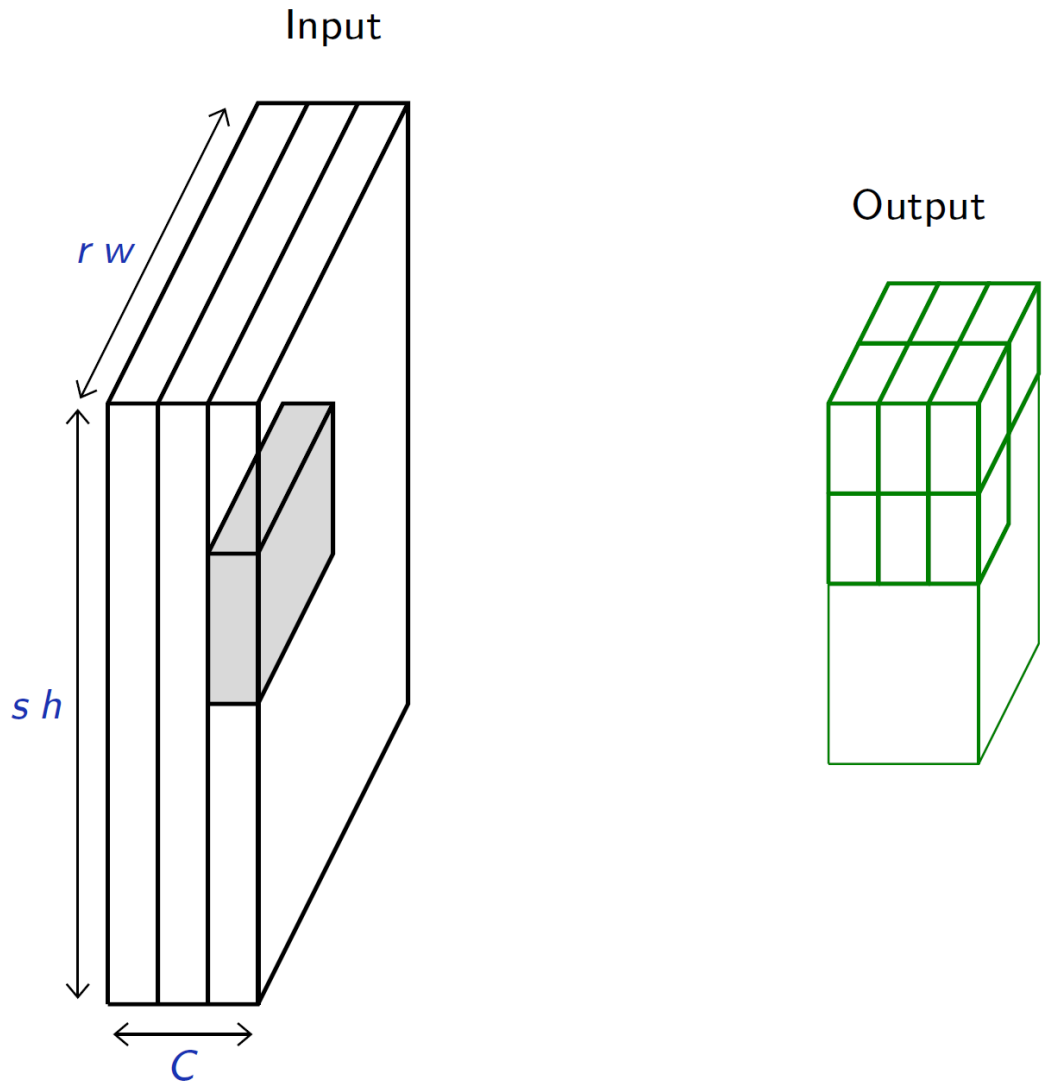
# Multi-channel Pooling



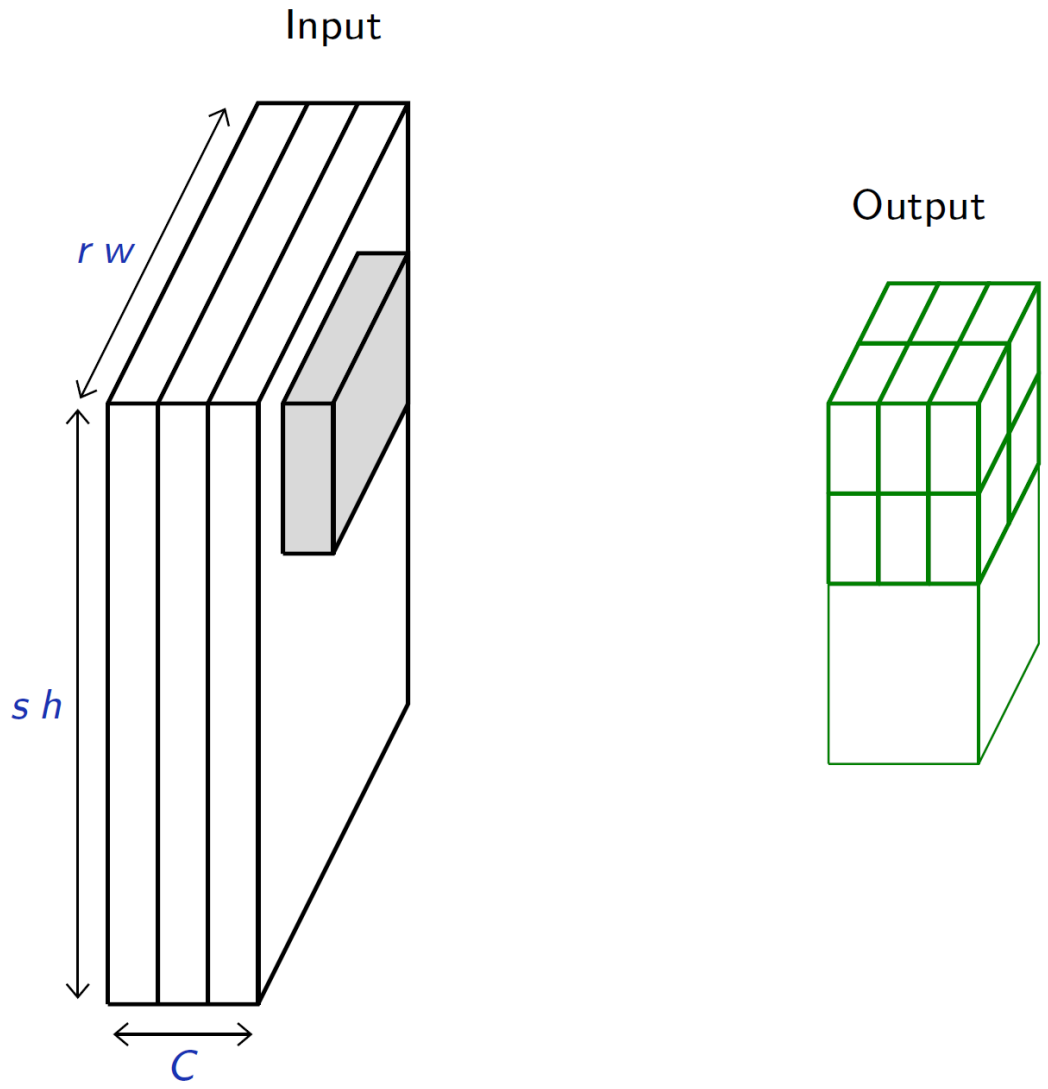
# Multi-channel Pooling



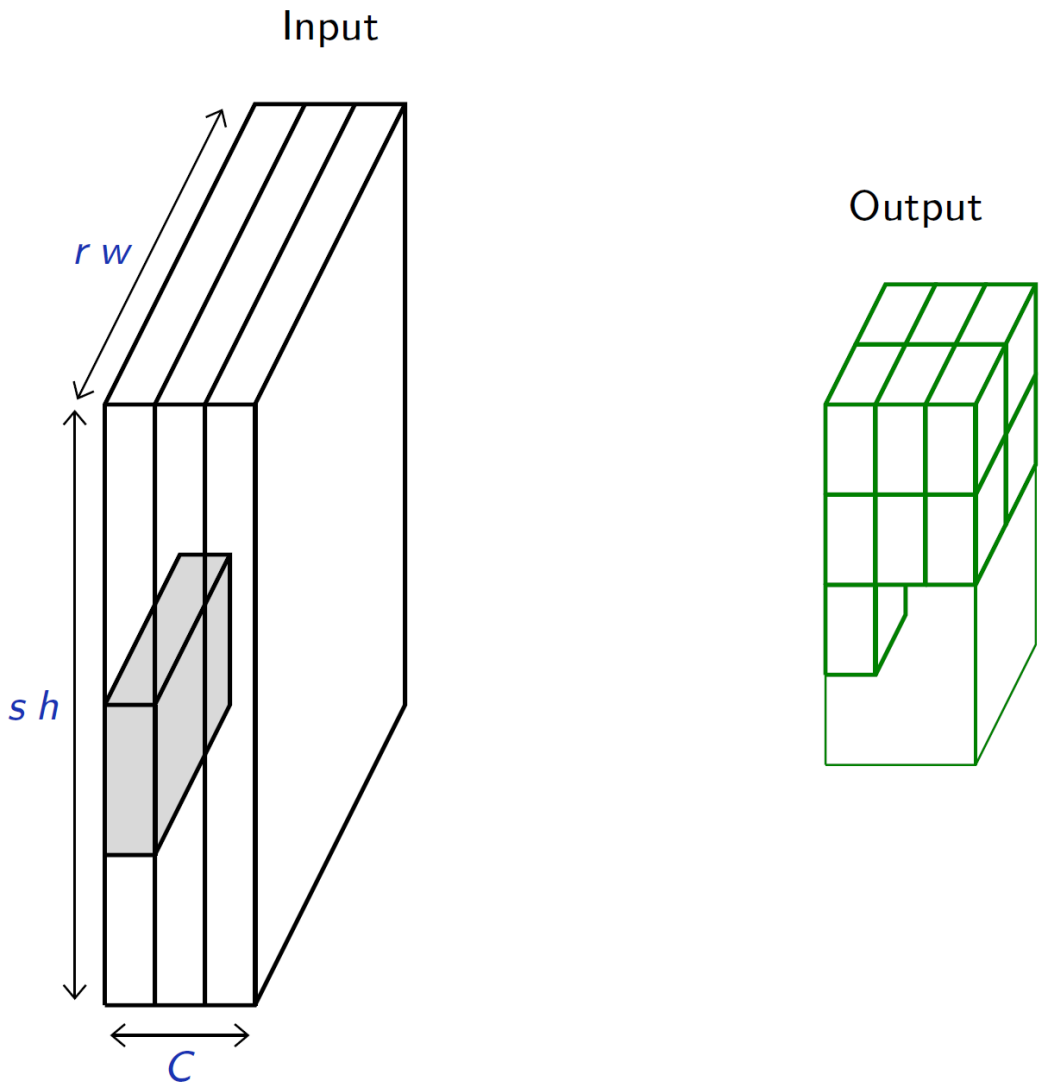
# Multi-channel Pooling



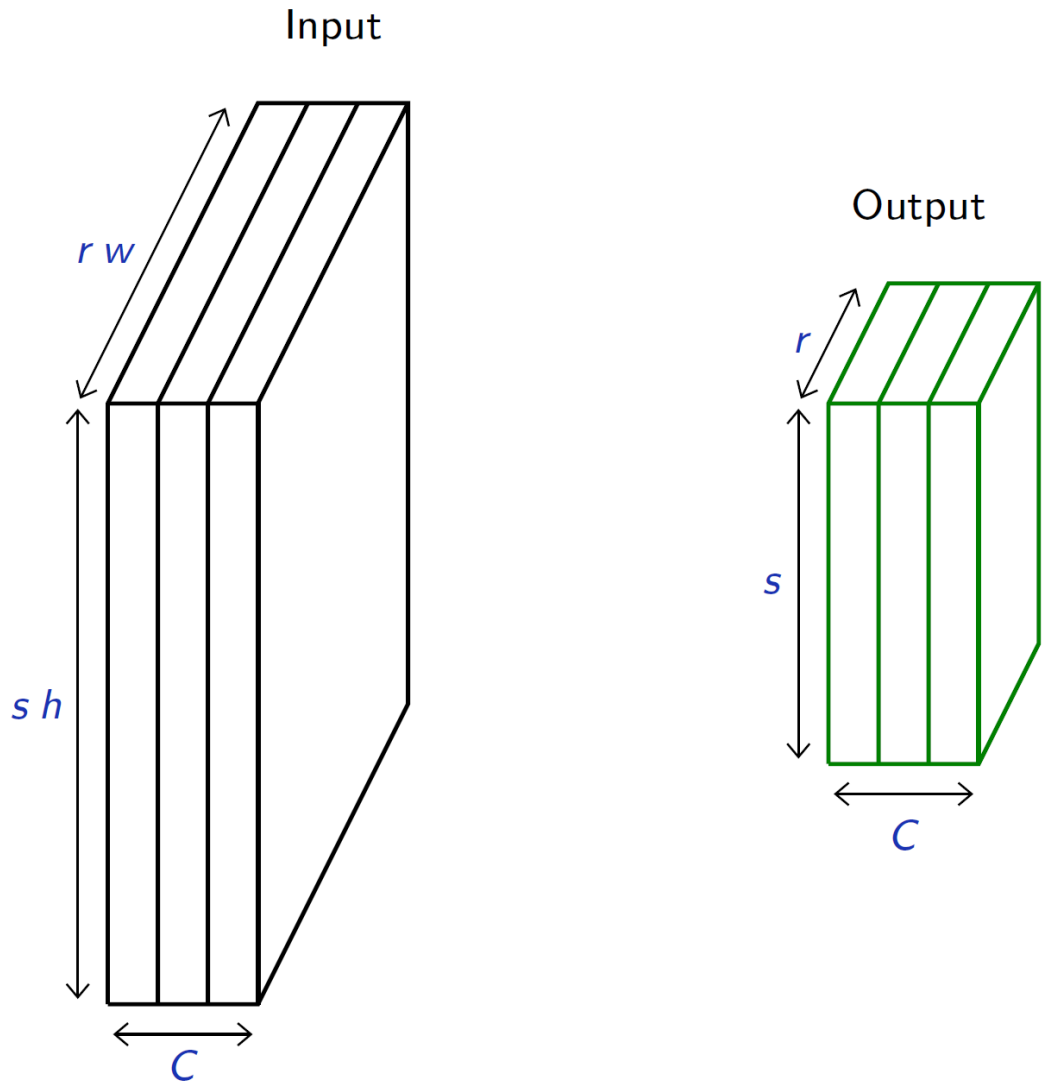
# Multi-channel Pooling



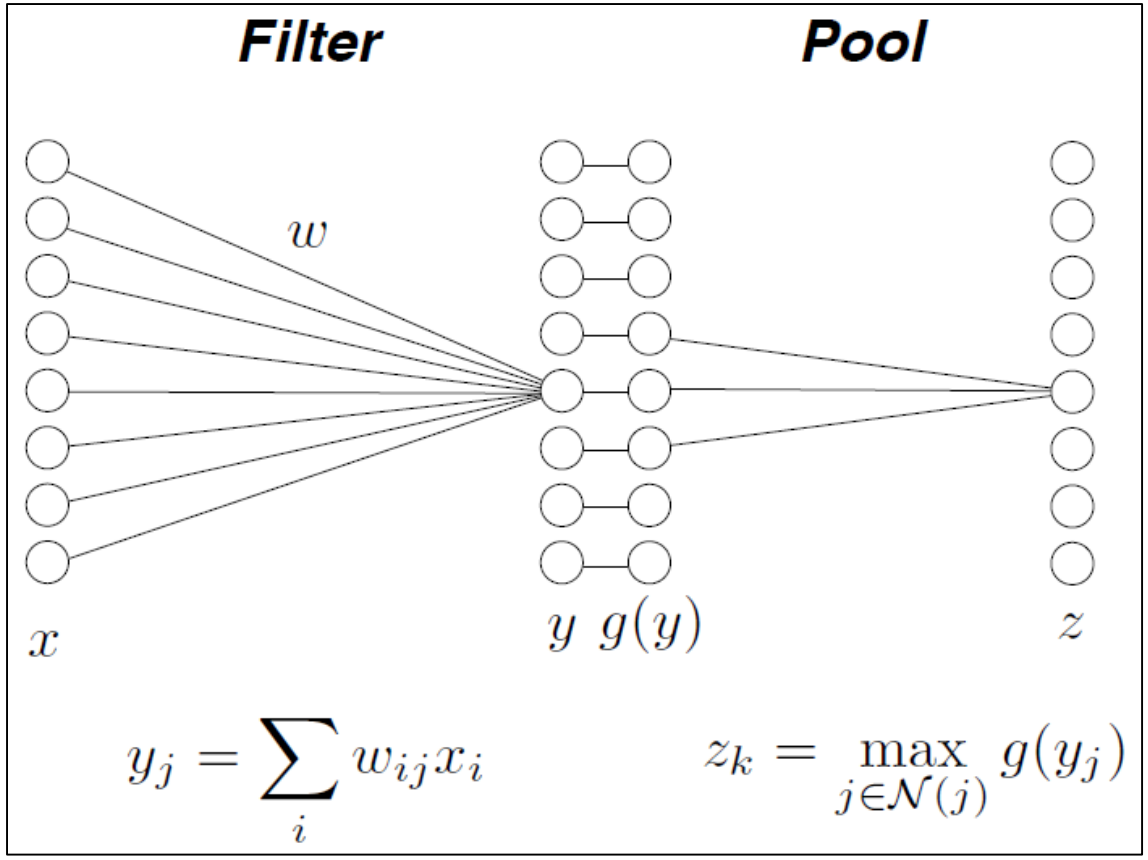
# Multi-channel Pooling



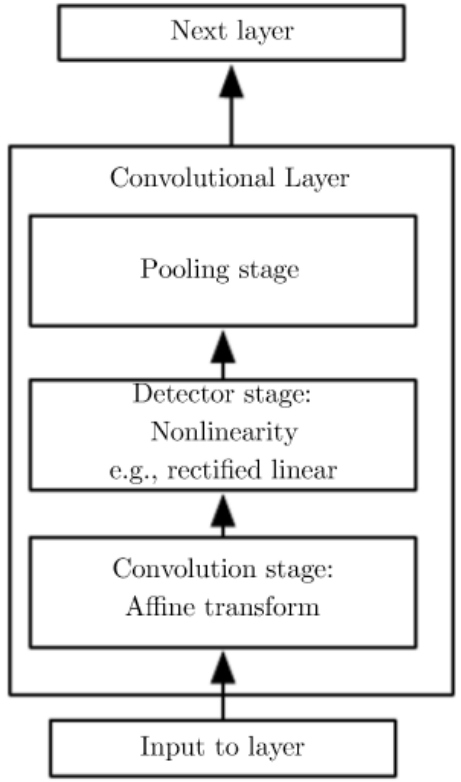
# Multi-channel Pooling



# Inside the Convolution Layer Block




Conv blocks

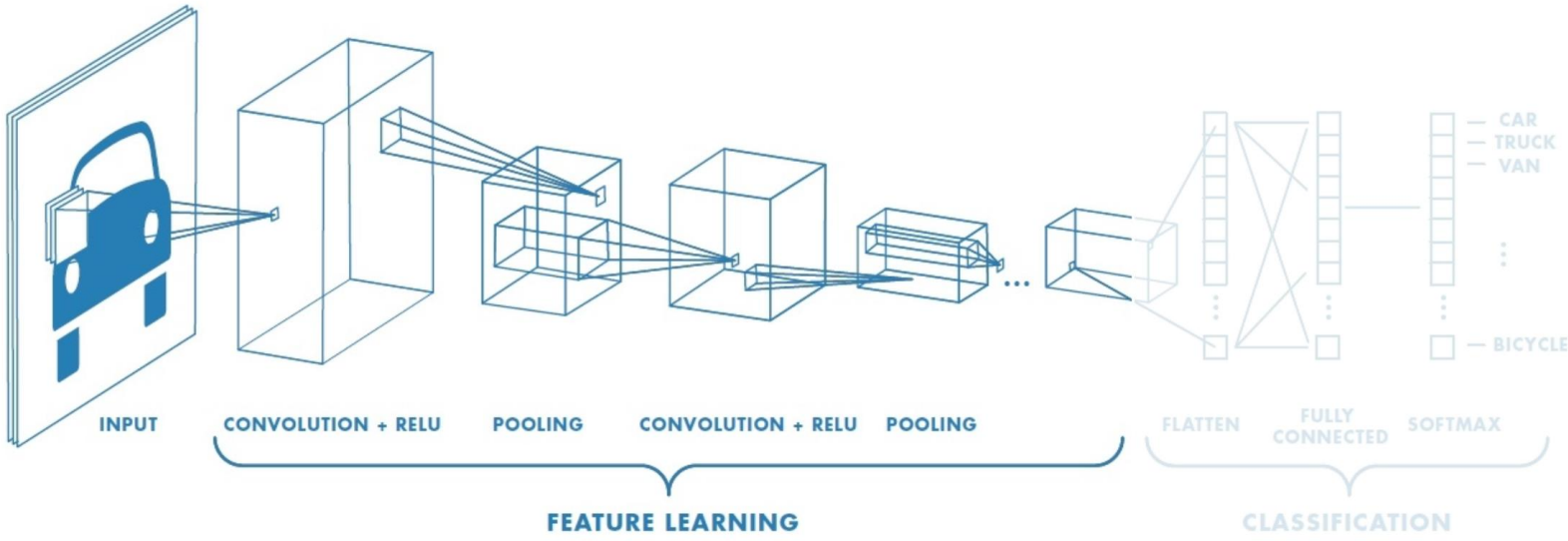


# Classic ConvNet Architecture

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

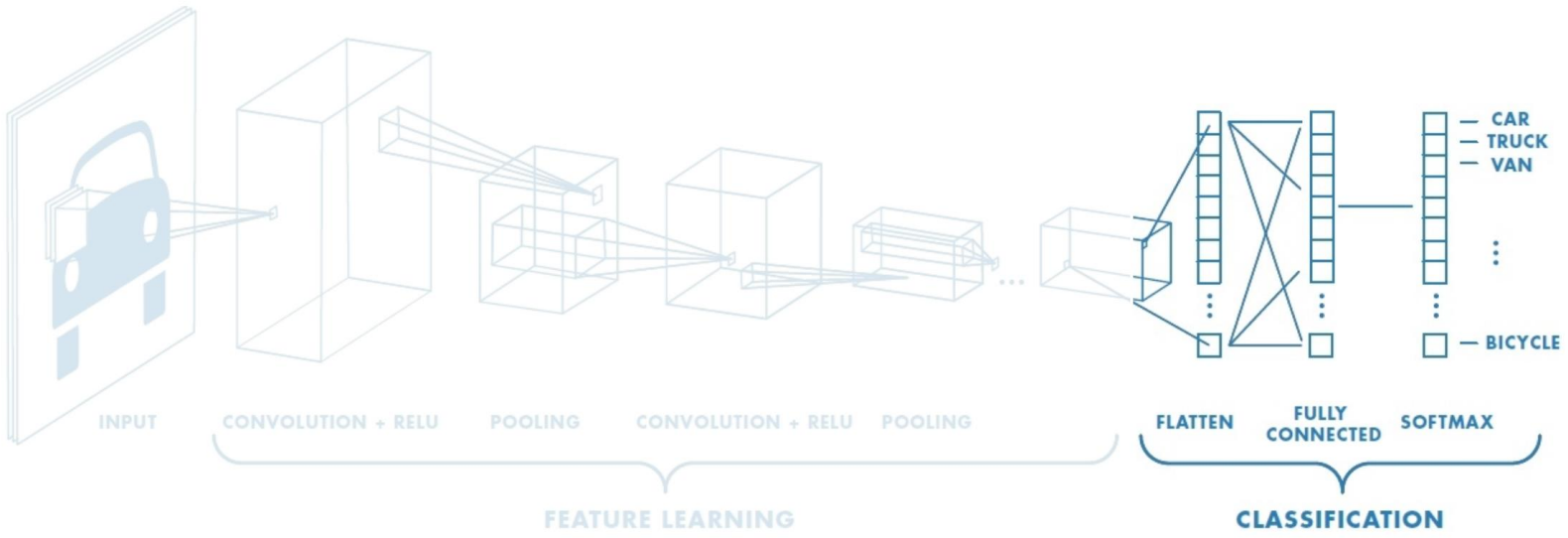
# CNNs for Classification: Feature Learning

- 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



# CNNs for Classification: Class Probabilities

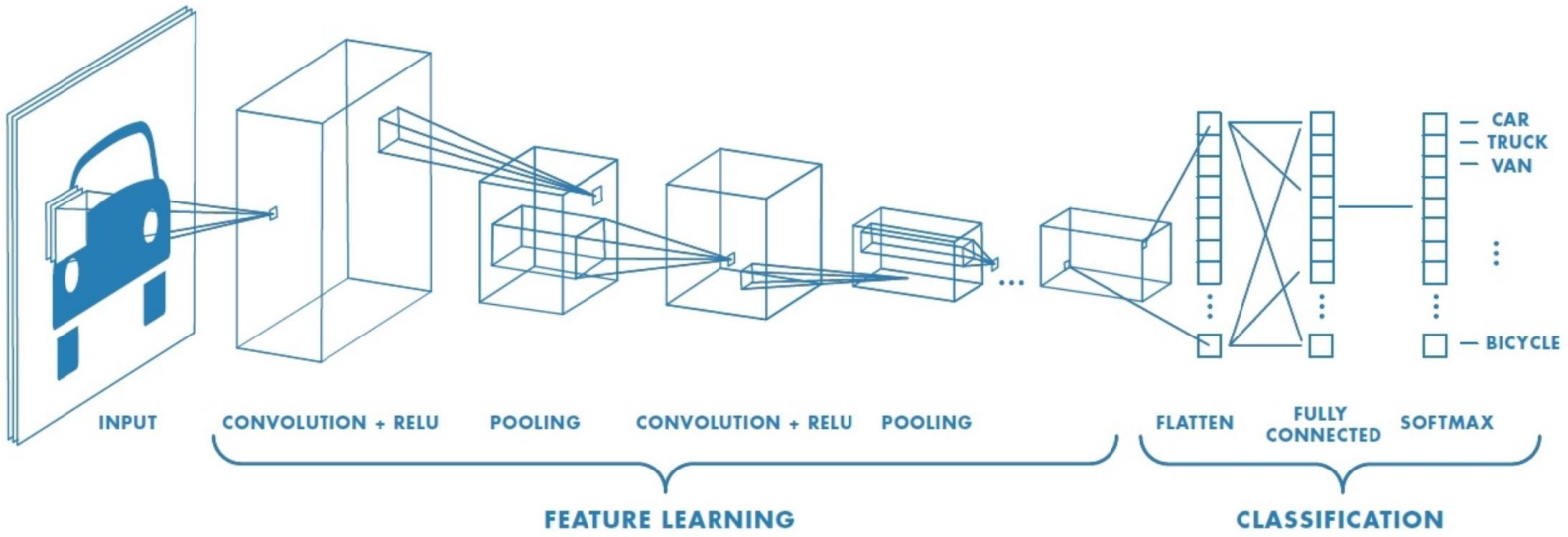
- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as probability of image belonging to a particular class




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

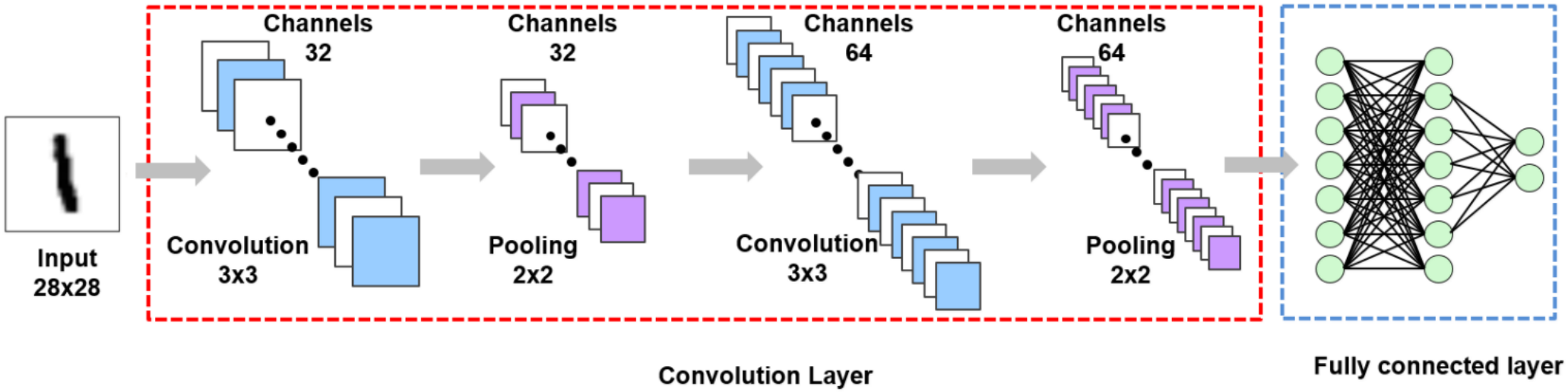
# CNNs: Training with Backpropagation

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



# Lab: CNN with TensorFlow

- MNIST example
- To classify handwritten digits



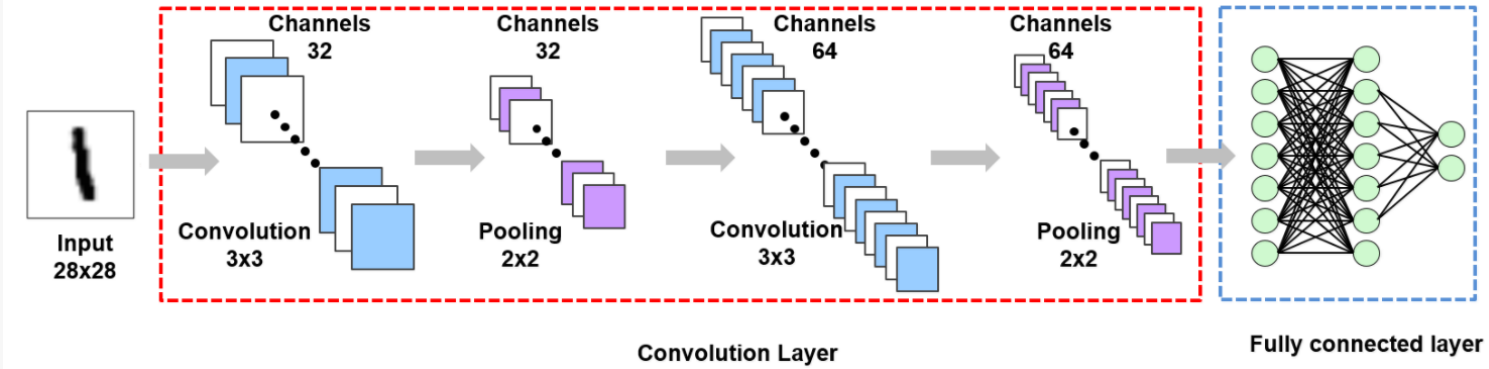
# CNN Structure

```
# 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
```

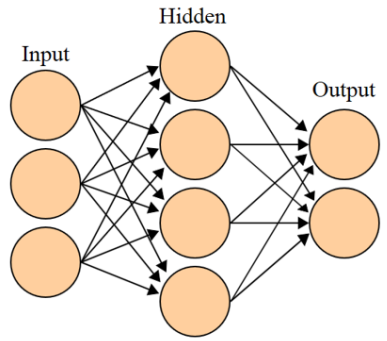


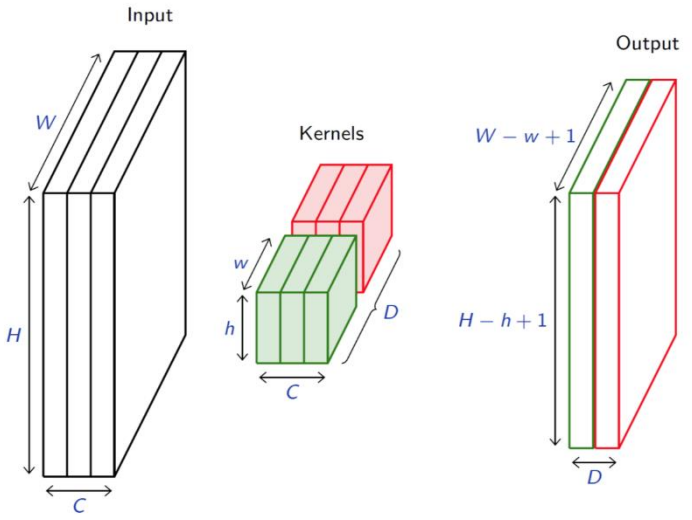
# Weights, Biases and Placeholder

```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])
```

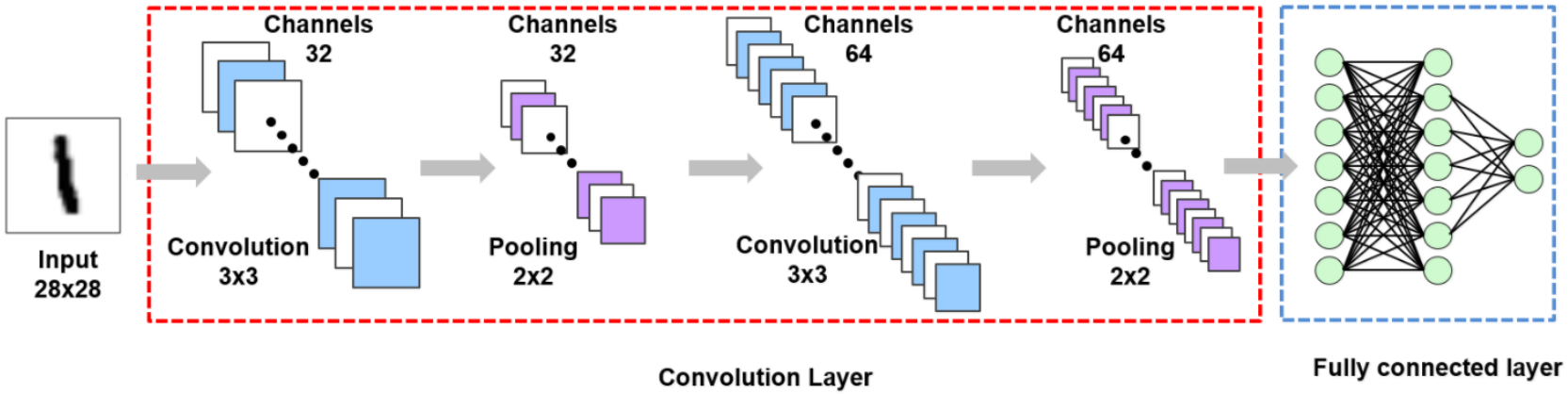


# Build a Model

- 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 apply 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%
```

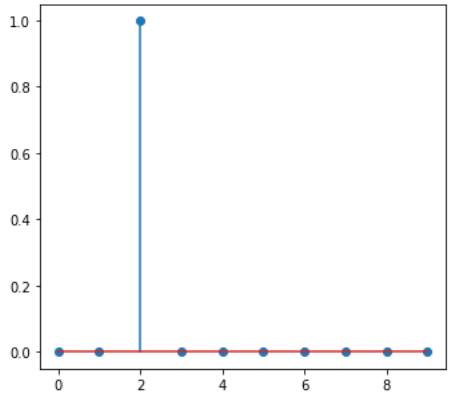
# Test or Evaluation

```
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