



Deep Learning

COMP 6721 Introduction of AI

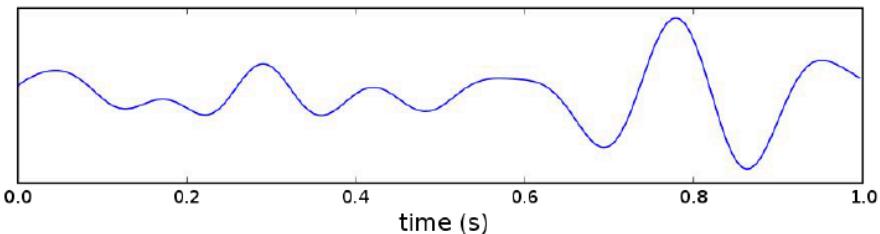
Russell & Norvig – Chapter 23.1 + 23.2 + 23.3

Topics

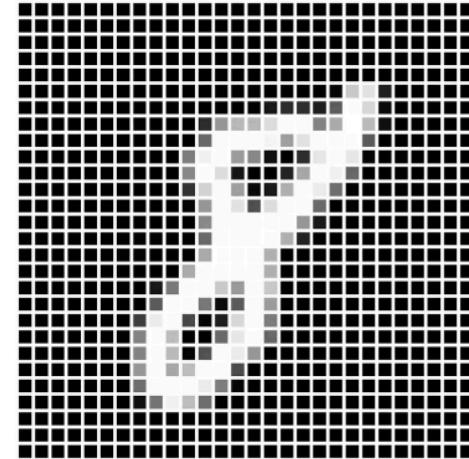
- ▶ CNN
- ▶ Image Representation
- ▶ Gradient Descent in CNNs

CNNs

- Convolutional neural networks, or CNNs, are a special type of neural network designed for processing data with a grid-like structure. These can be, for example:



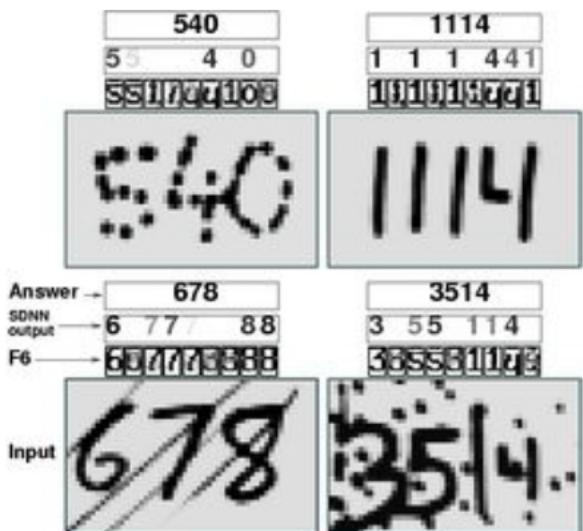
1D grids, e.g. time-series data taken at regular intervals



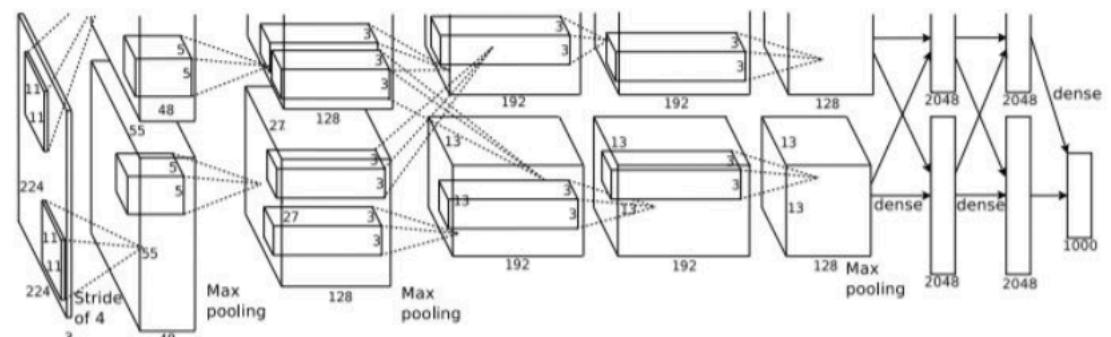
2D grids, e.g. images composed of pixels with intensity values

CNNs

- ▶ CNNs were some of the first deep learning models to be shown to generalize well to “useful” tasks.
- ▶ Modern interest in CNNs came about when Alex Krizhevsky et al. won the 2012 **ImageNet** object recognition challenge using **AlexNet**.

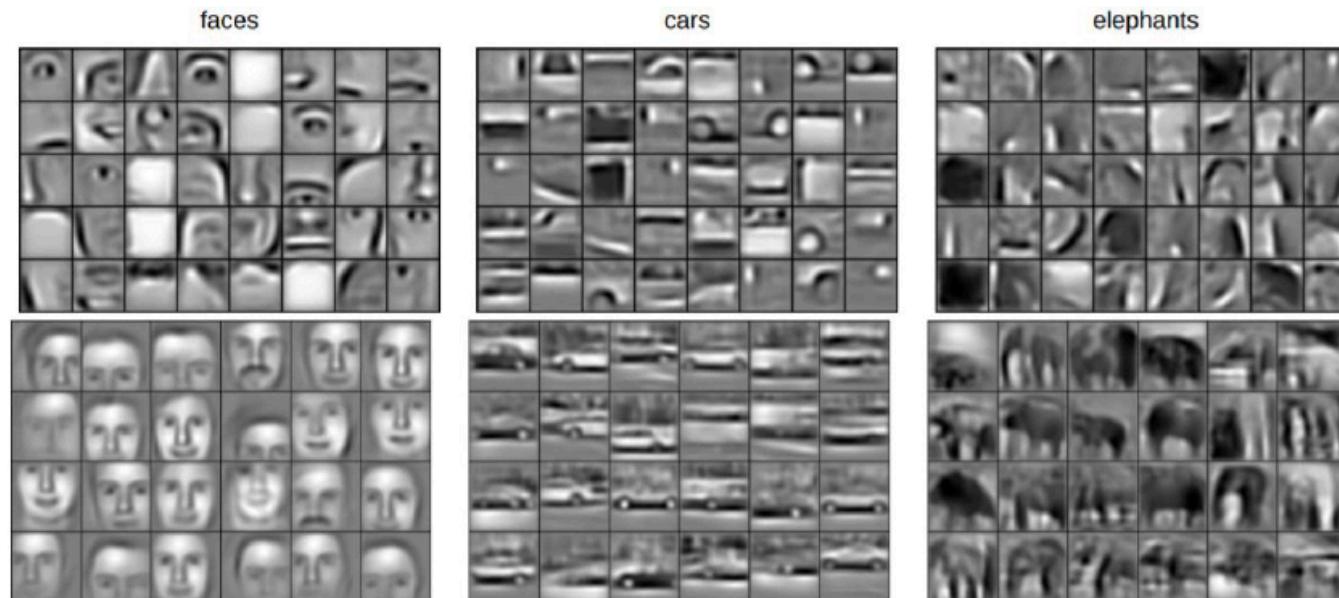


IMAGENET



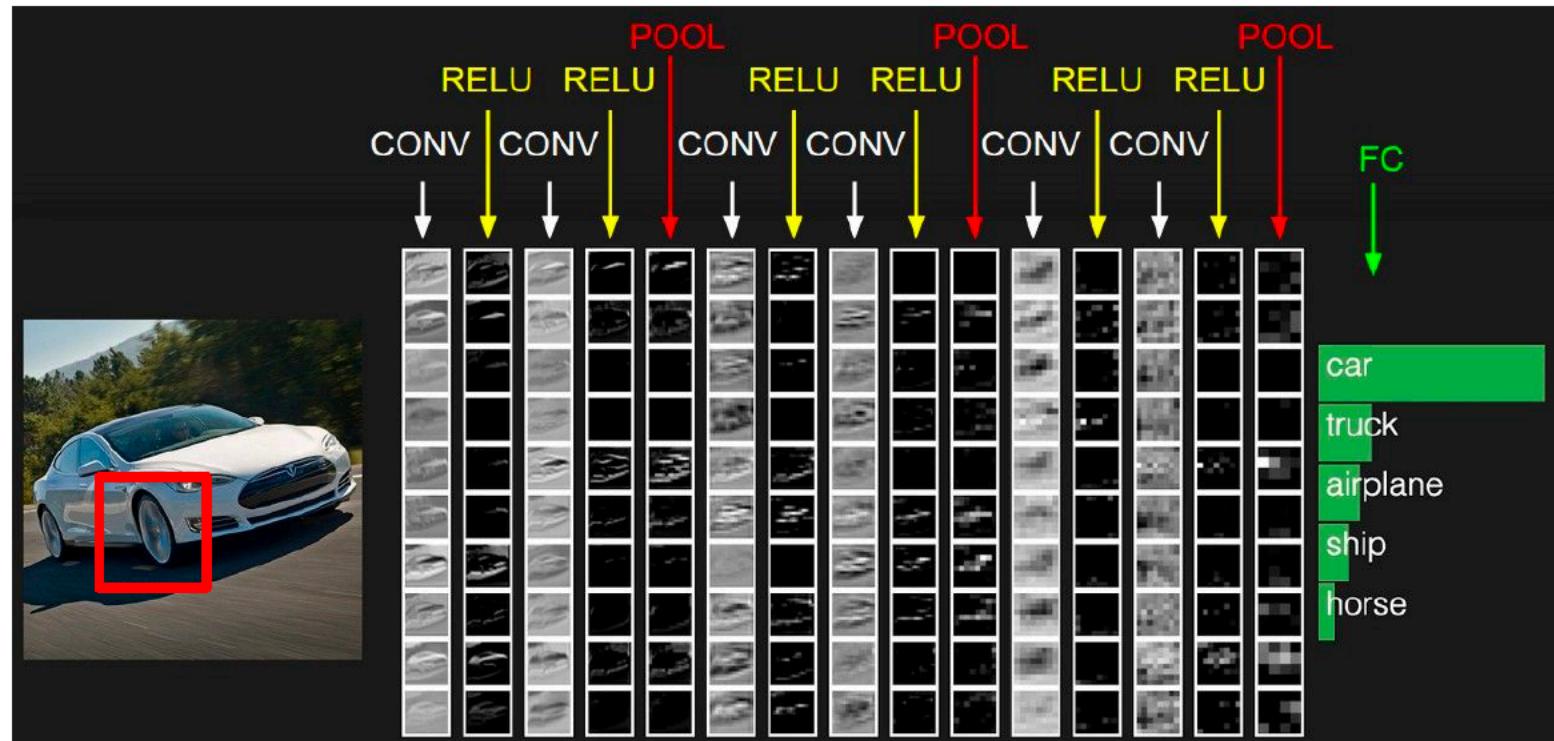
CNNs

- ▶ CNNs were introduced to encourage neural networks to focus on local features, or “patches”, in images.
- ▶ Think of a detector that scans images looking for “eye-like” and “nose-like” features to detect a human.



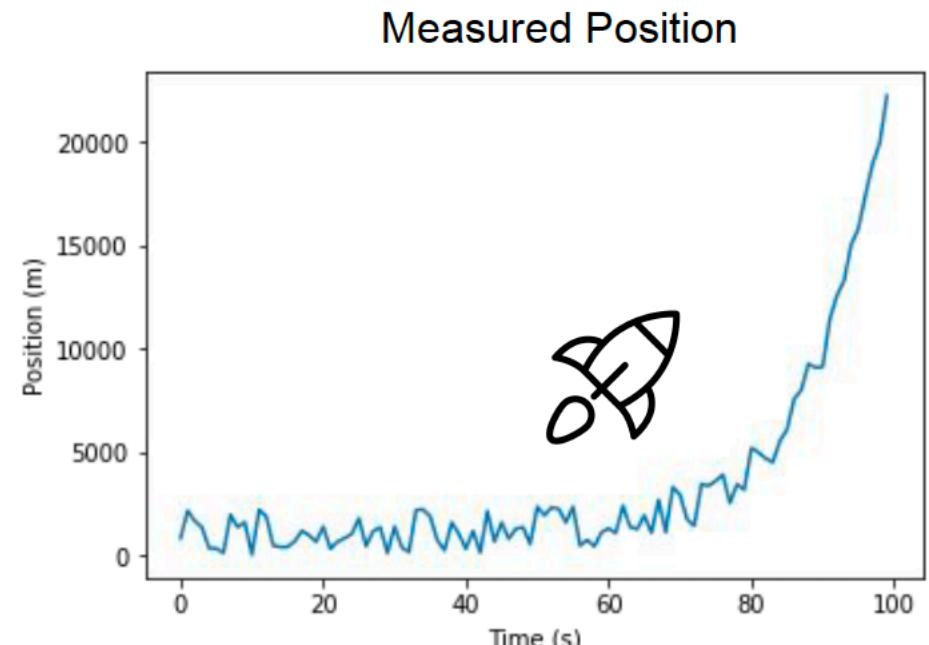
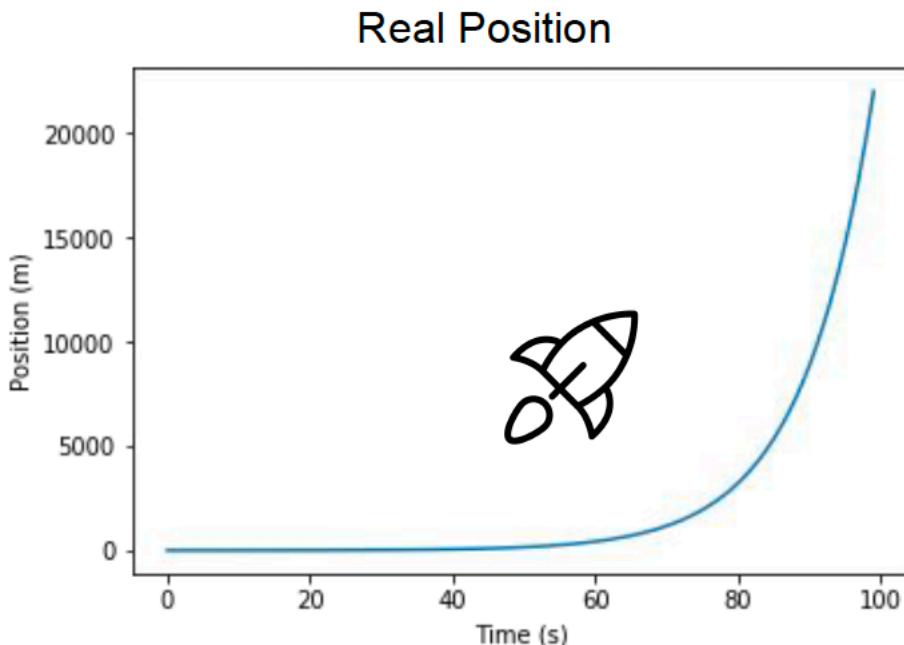
CNNs

- ▶ For example, features can be extracted at different local regions to predict whether a car is present in an image or not.



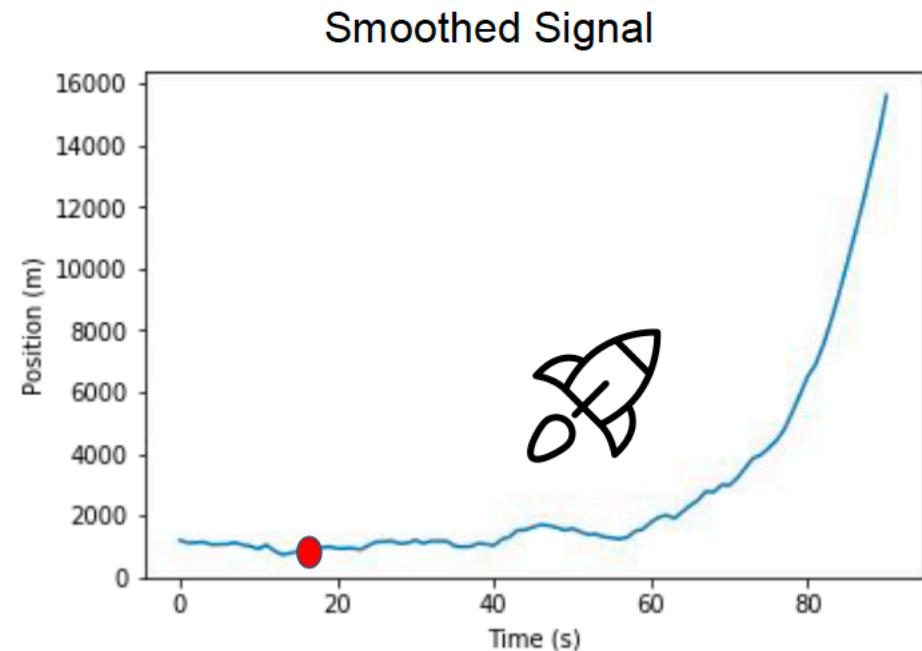
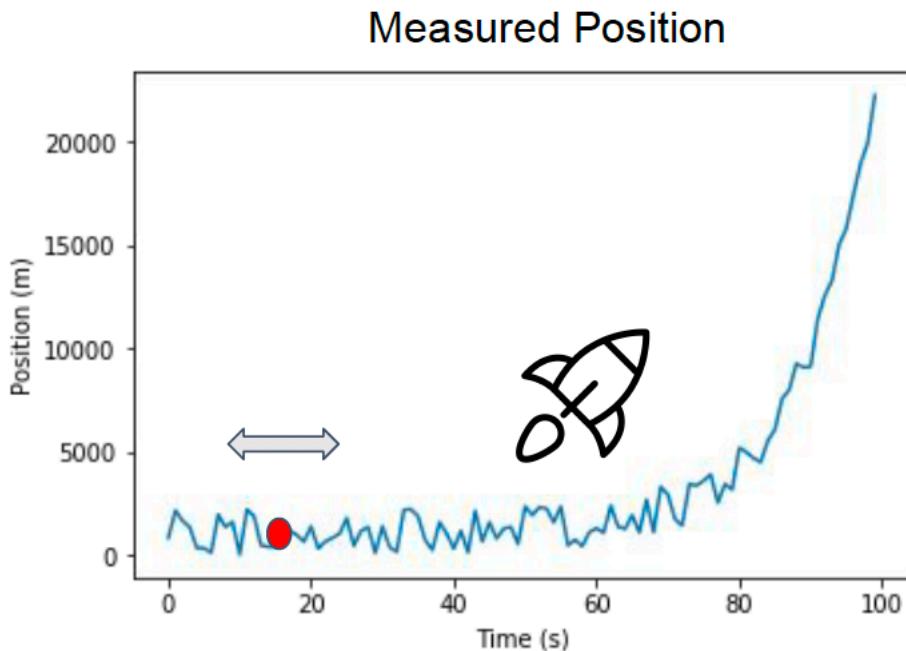
Convolution - Intuition

- ▶ Suppose you are recording the position of a spaceship with a laser. However, you notice inherent noise in your measurements (radiation, temperature, etc.).



Convolution - Intuition

- A “simple” way to clean the obtained signal is to use a moving average. This is an example of smoothing, and can easily be defined using a convolution.



Convolution - Intuition

- ▶ Convolutions are a special type of mathematical operation. They are the foundation behind CNNs. Mathematically, a convolution is defined using the $*$ operator:

$$s(t) = (x * w)(t)$$

- ▶ where $s(t)$ is said to be the convolution of $x(t)$ with $w(t)$ and t is typically time. In the context of CNNs, x is typically referred to as the input, w as the kernel and s as the feature map.
- ▶ More formally:

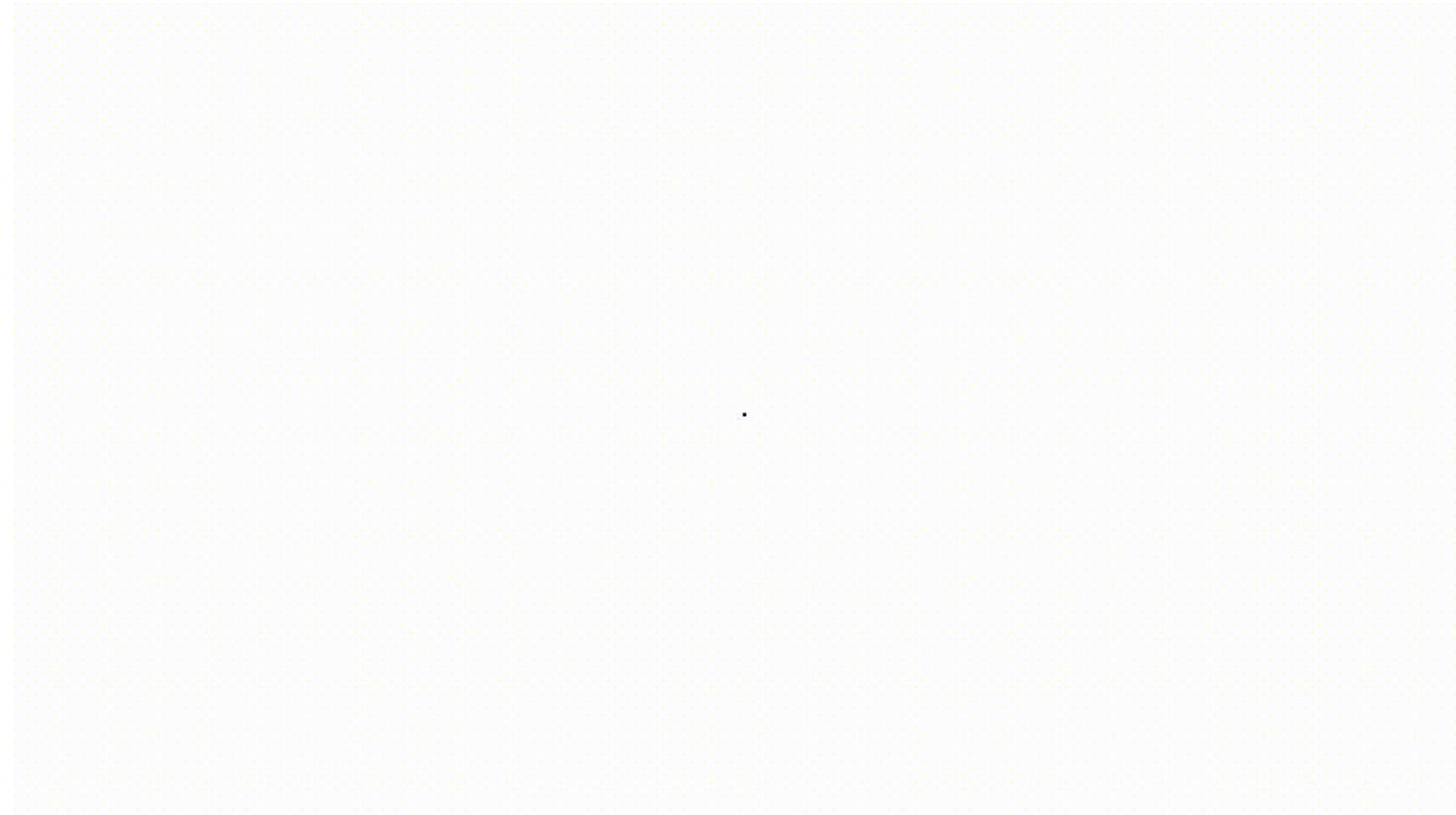
$$s(t) = \int x(a)w(t-a)da.$$

Convolution for a continuous signal

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Convolution for a discrete signal

1D Convolution



1D Convolution

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$



$w(a)$ ← Kernel



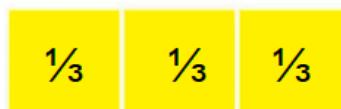
$x(a)$ ← Input



$s(t)$ ← Output

1D Convolution

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$



$w(a)$ ← Kernel



$x(a)$ ← Input



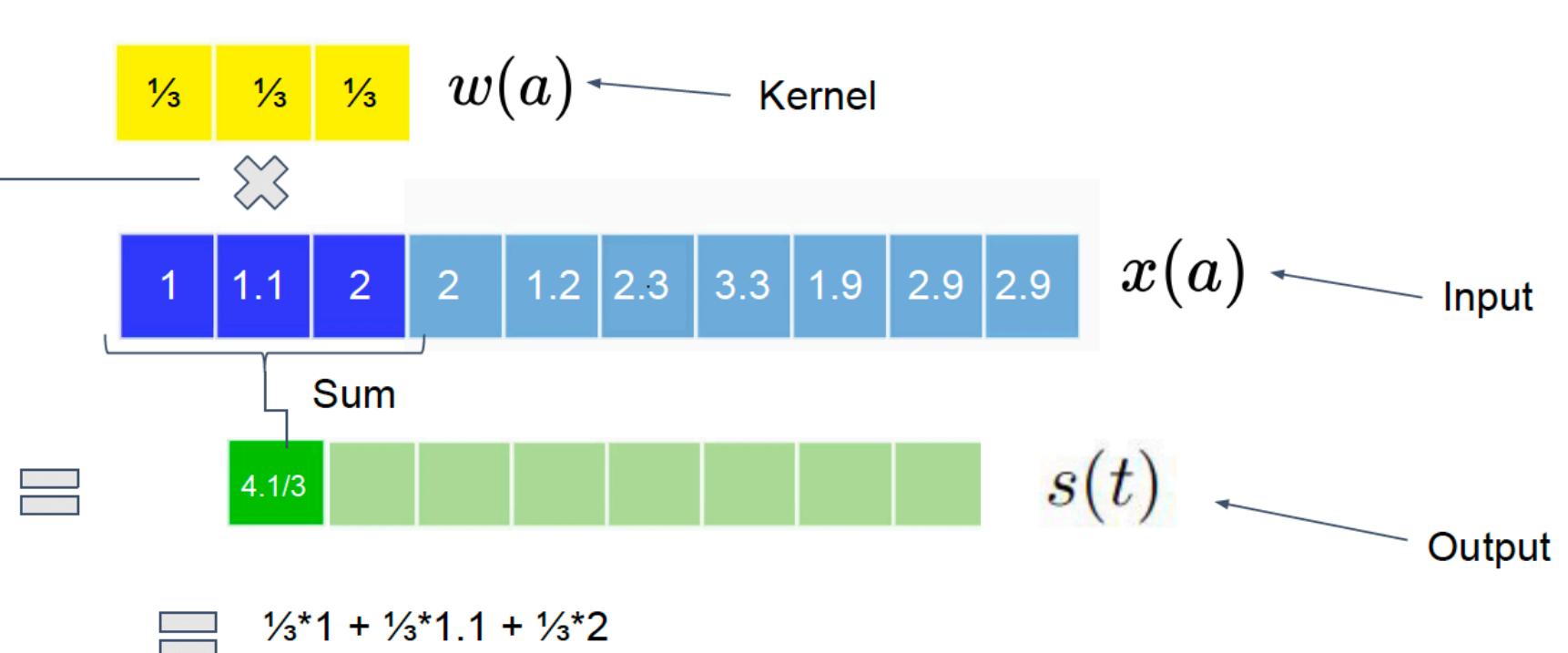
$s(t)$

← Output

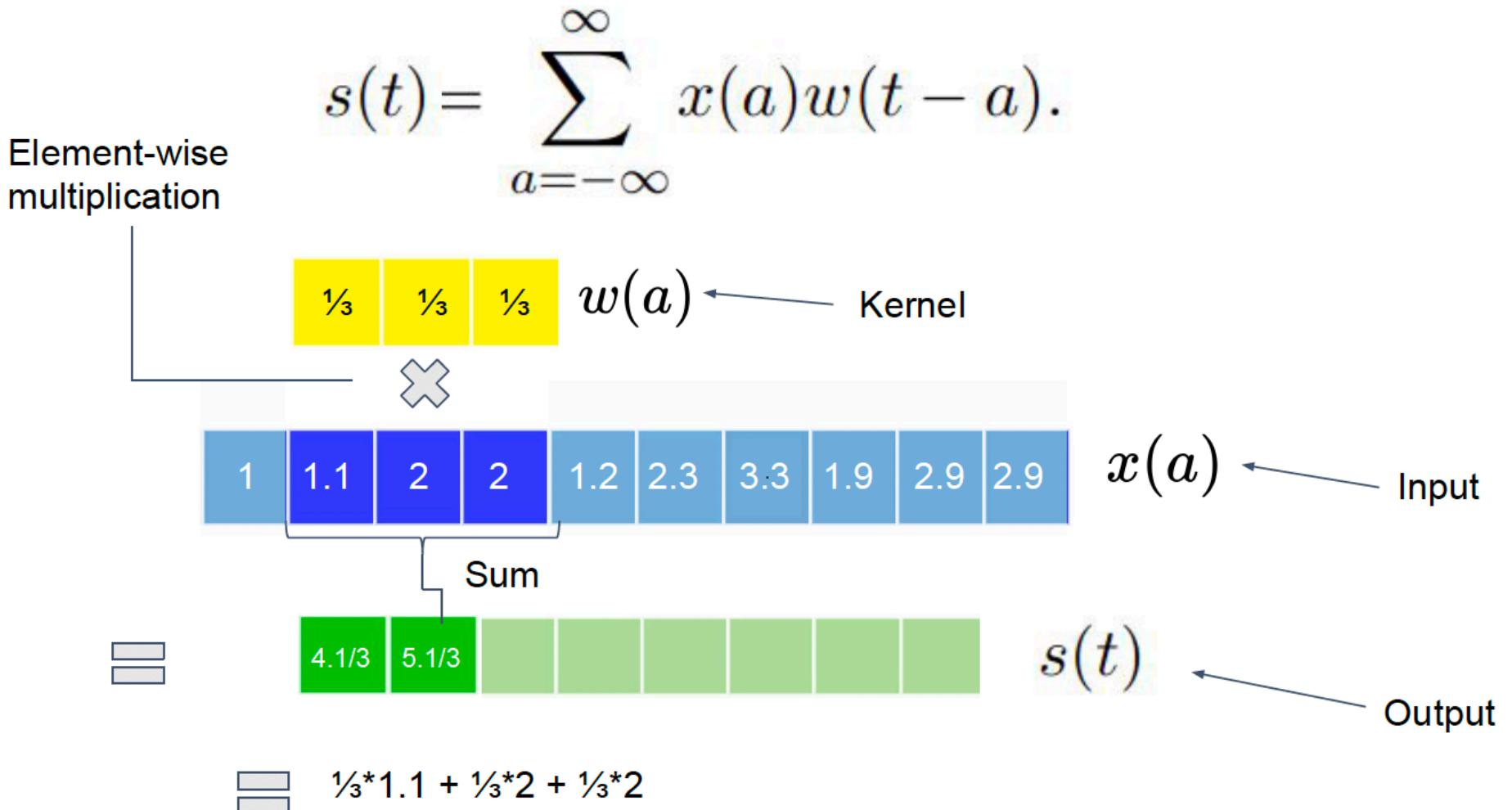
1D Convolution

逐元素乘法
Element-wise
multiplication

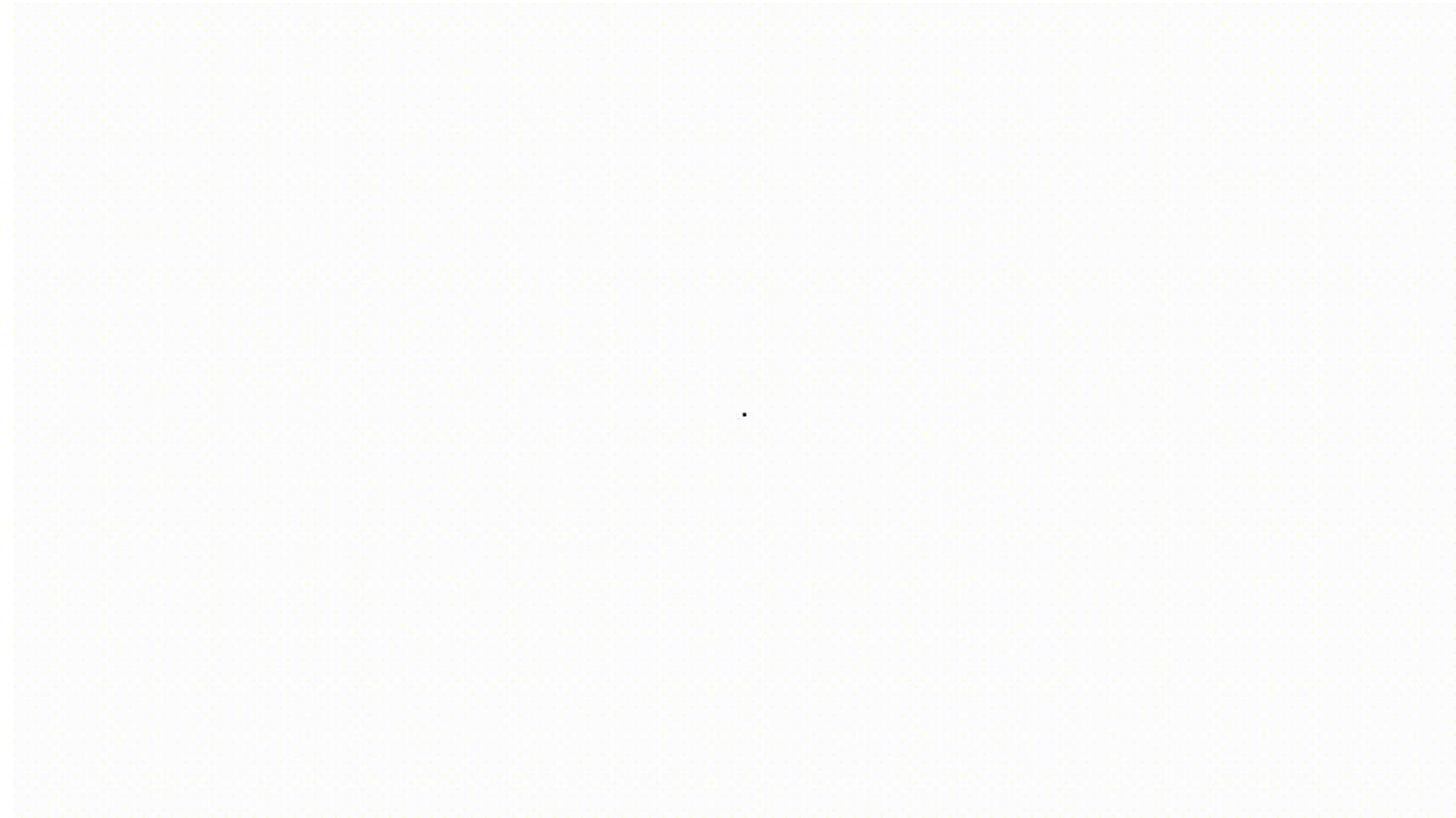
$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$



1D Convolution



1D Convolution

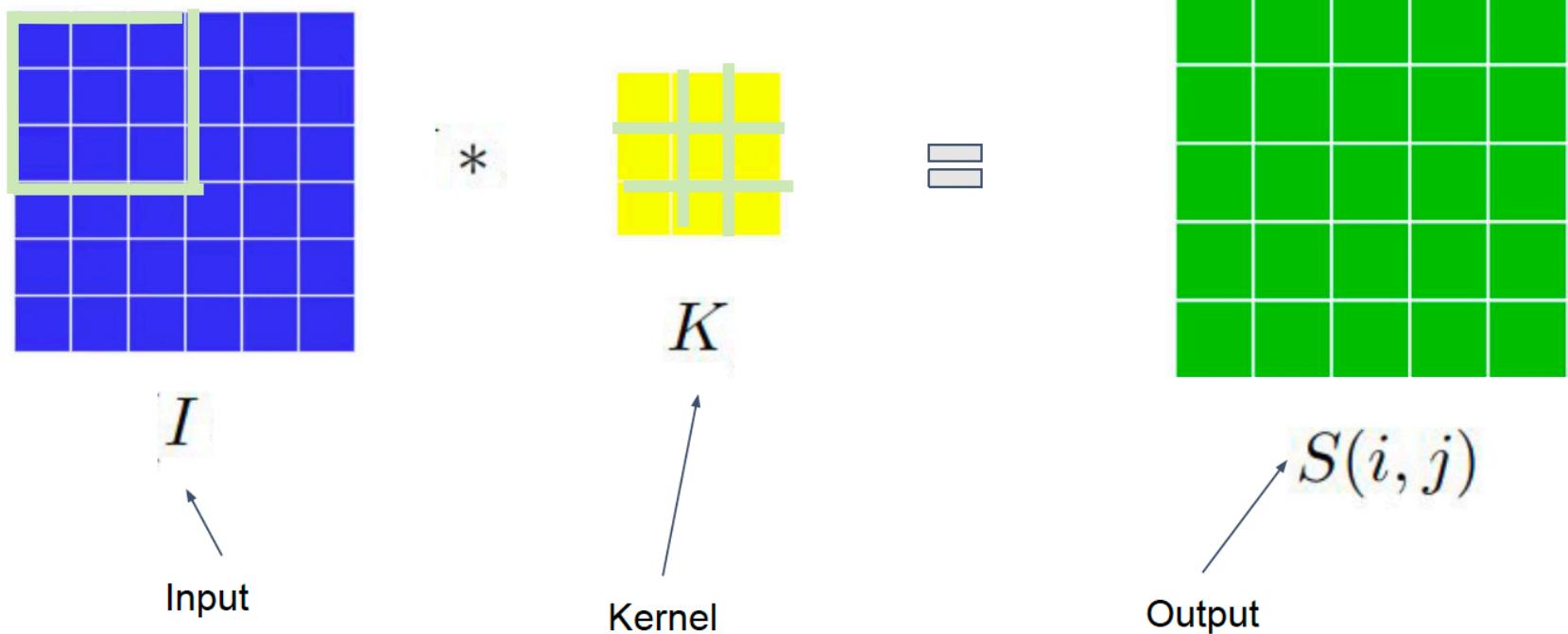


2D Convolution



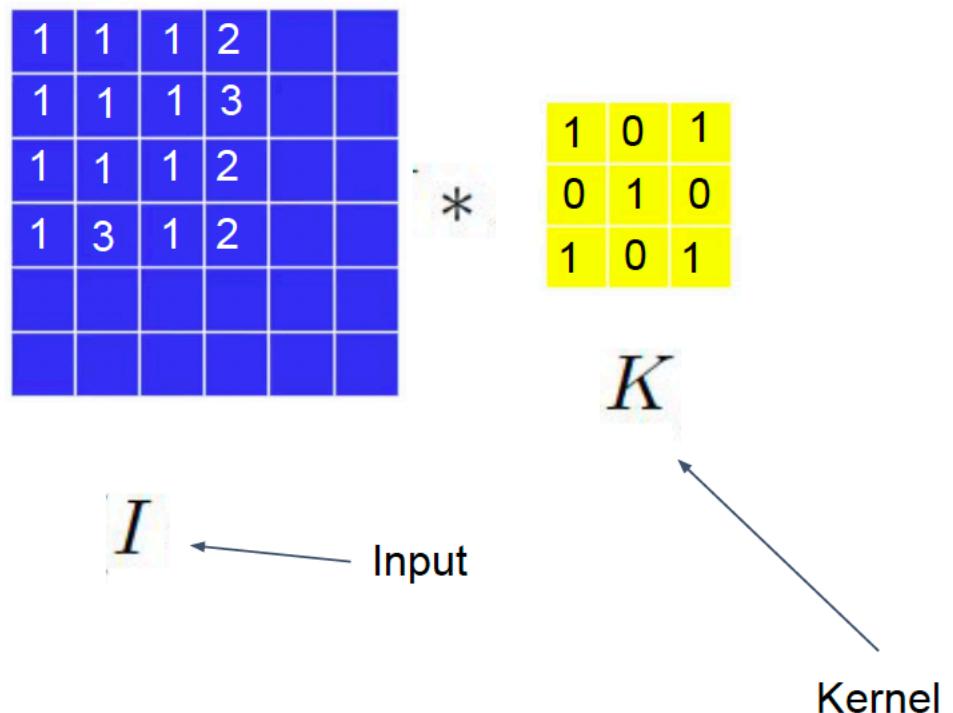
2D Convolution

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$



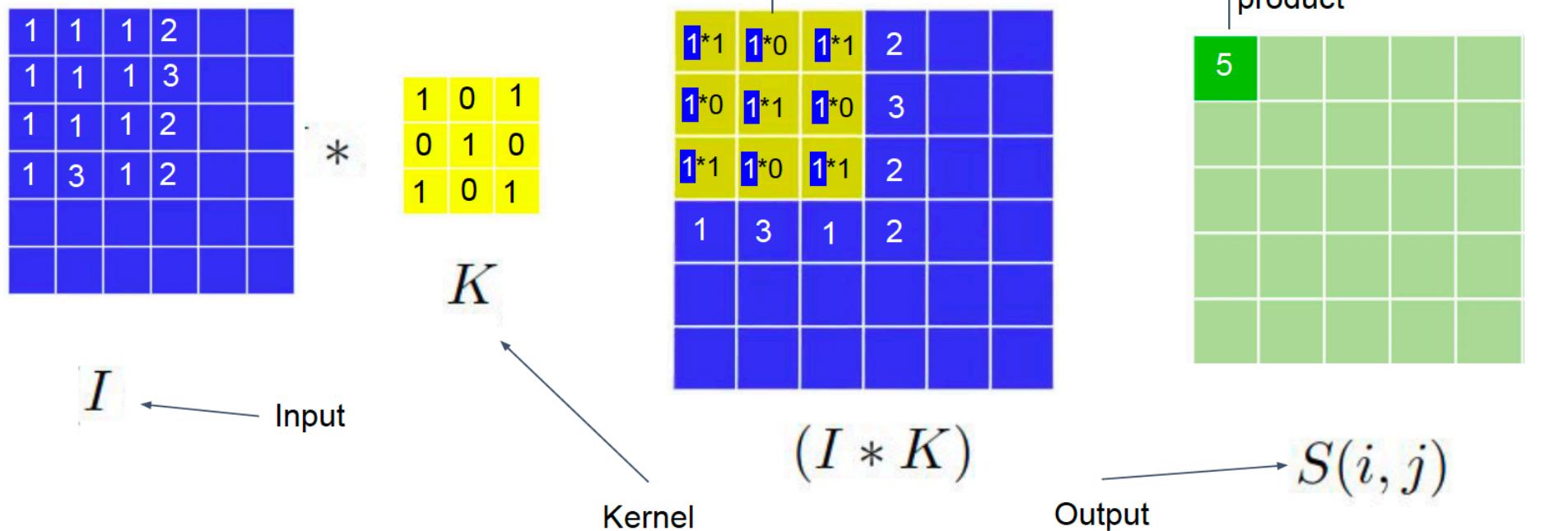
2D Convolution

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$



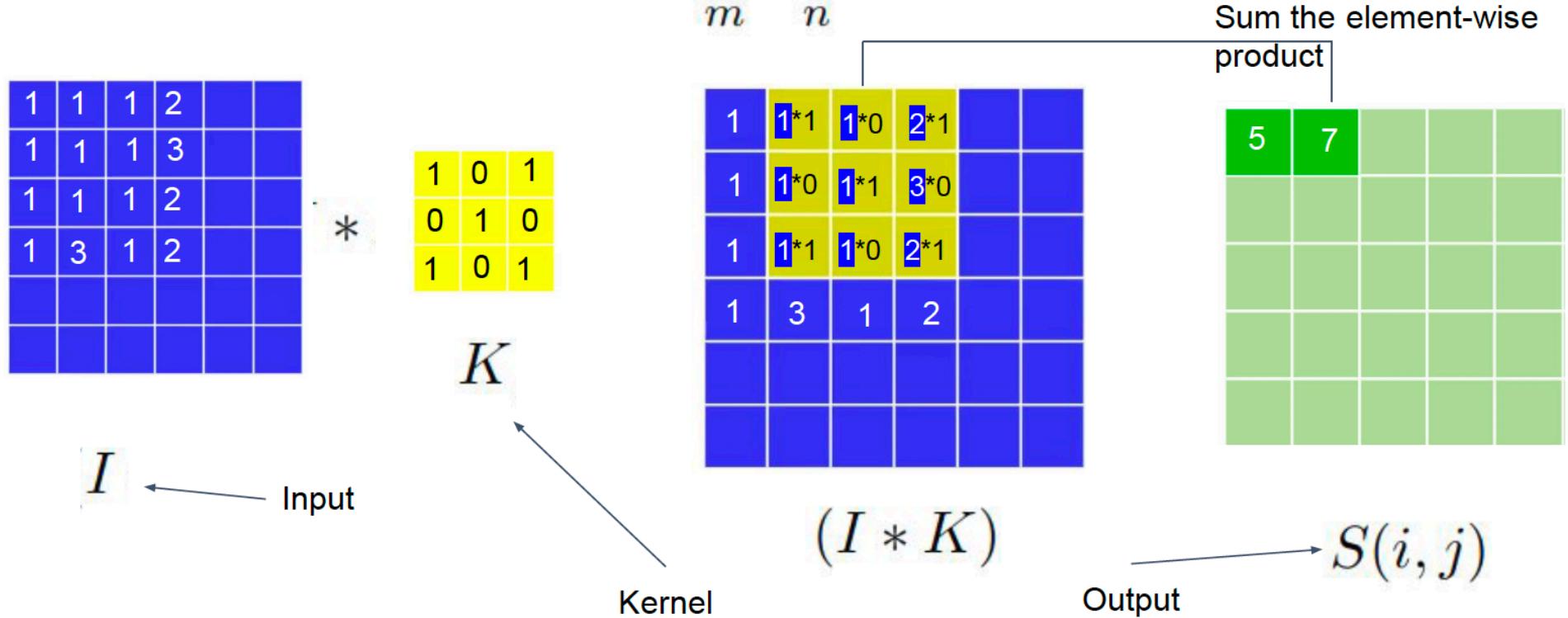
2D Convolution

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n).$$



2D Convolution

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n).$$

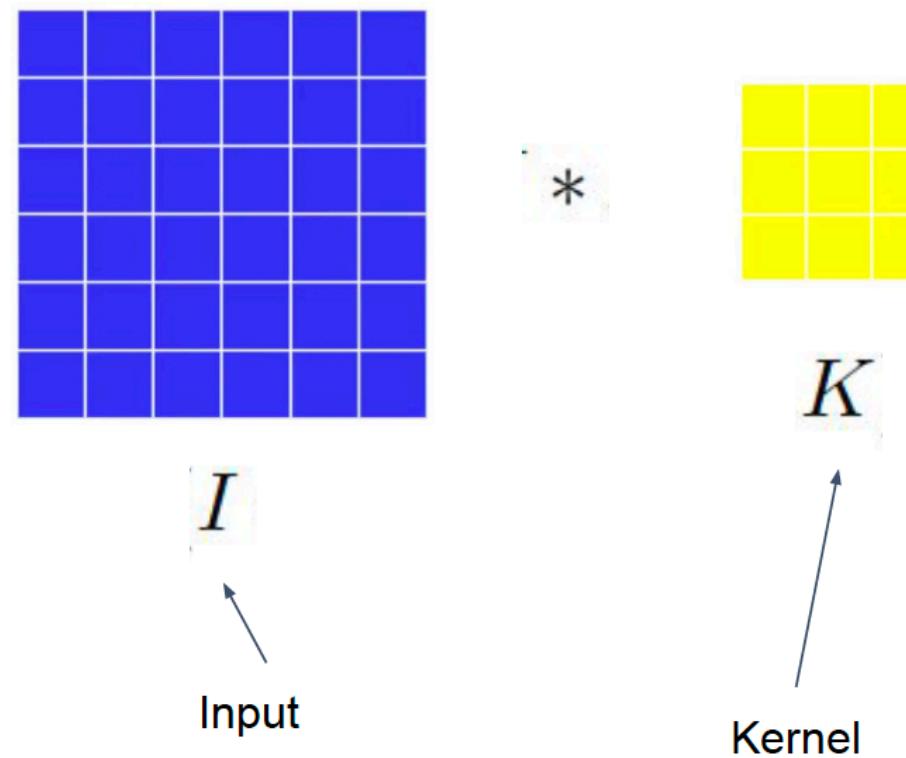


2D Convolution

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n)$$

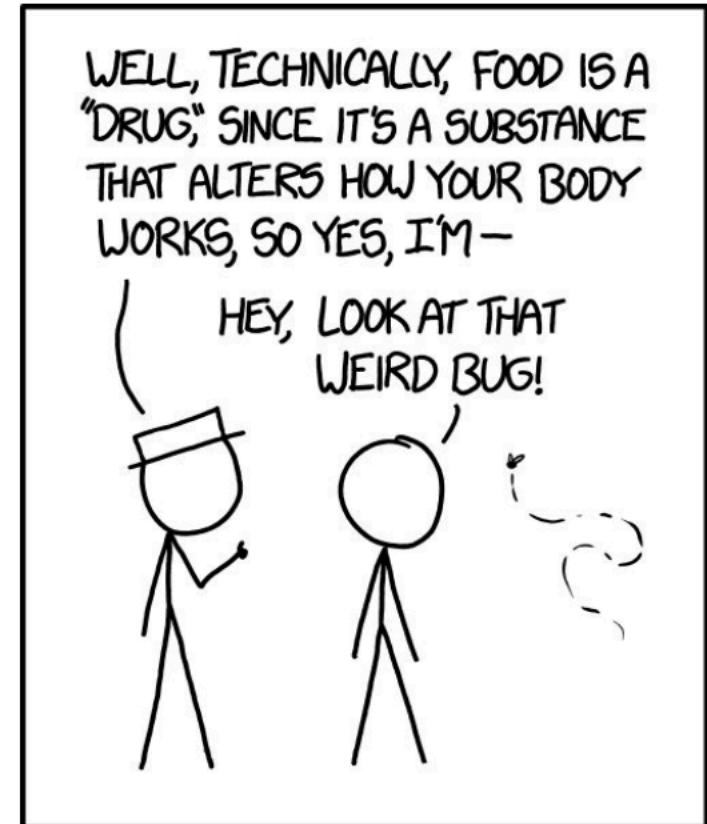
2D Convolution

The idea is that the network can **learn** the optimal parameters of the **kernel** to solve a given task.



Pedantic Timeout

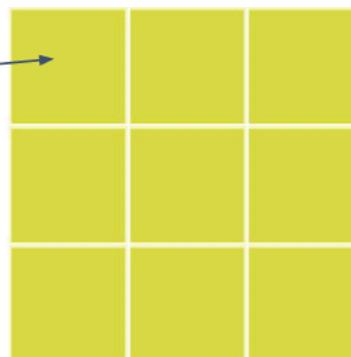
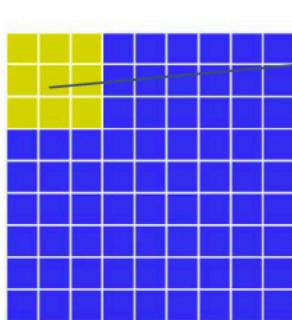
Technically, convolutions in neural networks are really implemented as **cross-correlations** (the kernels are not flipped as they should be).



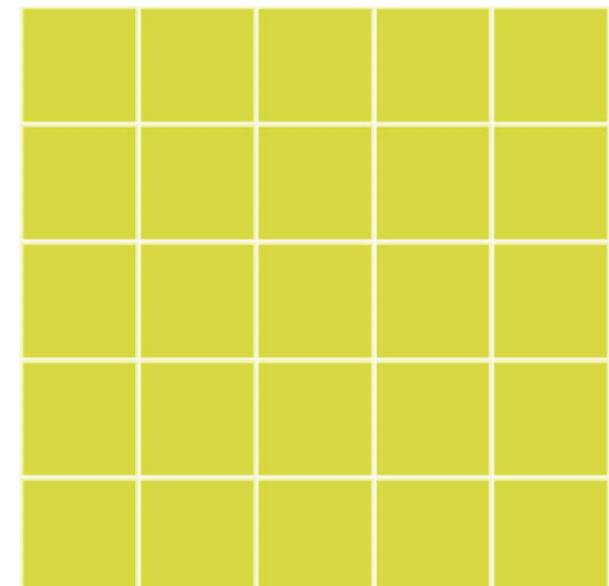
MY LIFE IMPROVED WHEN I REALIZED I COULD JUST IGNORE ANY SENTENCE THAT STARTED WITH "TECHNICALLY."

Spatial Extent

The size of the kernel, or filter, usually denoted F , references how many pixels the kernel spans. Kernels are typically square in size.



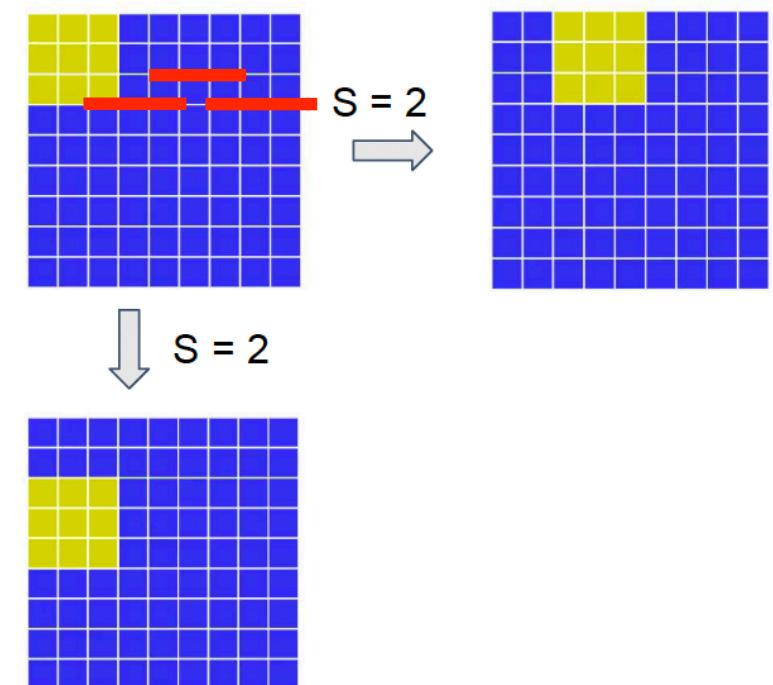
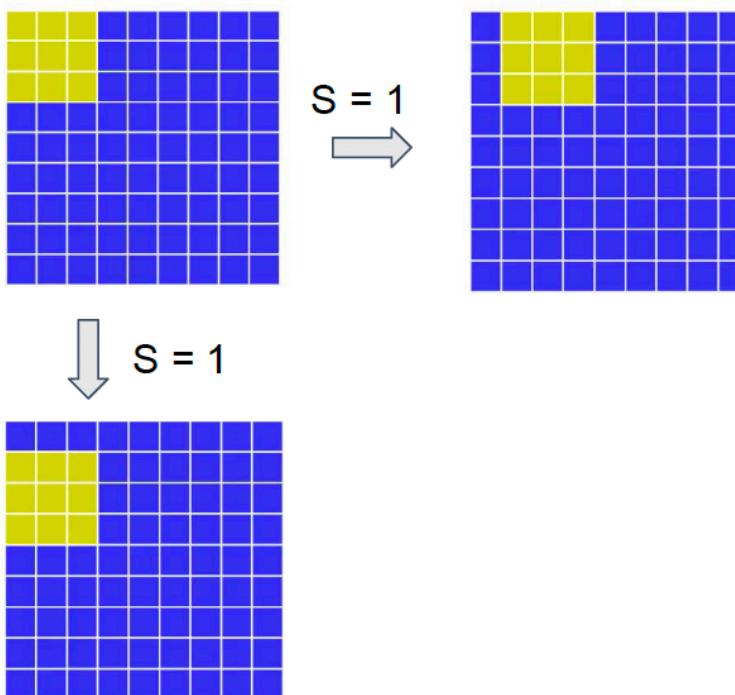
$F = 3$



$F = 5$

Stride

The stride, often denoted as S , represents the number of pixels to move in (x,y) before each operation in the convolution. Typical values used are either $S=1$ or $S=2$.



Stride

The stride, often denoted as S , represents the number of pixels to move in (x,y) before each operation in the convolution. Typical values used are either $S=1$ or $S=2$.



Stride = 1 in both x and y Stride = 2 in both x and y

Stride

The stride, often denoted as S , represents the number of pixels to move in (x,y) before each operation in the convolution. Typical values used are either $S=1$ or $S=2$.

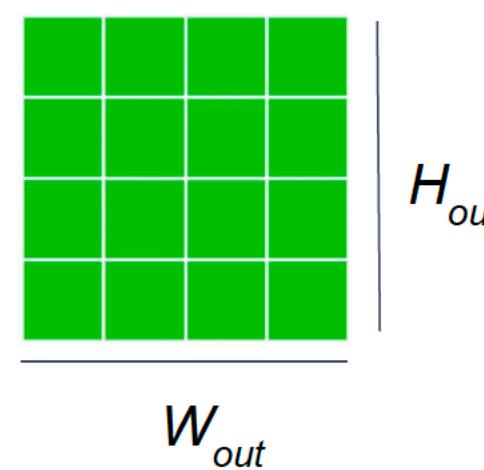
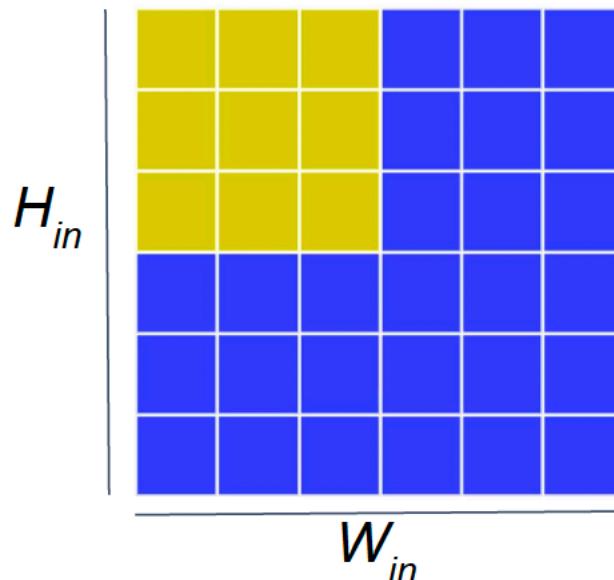
Stride = 2 in both x and y

2D Convolution Arithmetic

In general, if we have an input of size $W_{in} \times H_{in}$, a stride S and a spatial extent F, our output will have dimensions

$$W_{out} = (W_{in} - F) / S + 1$$

$$H_{out} = (H_{in} - F) / S + 1$$



In this example:

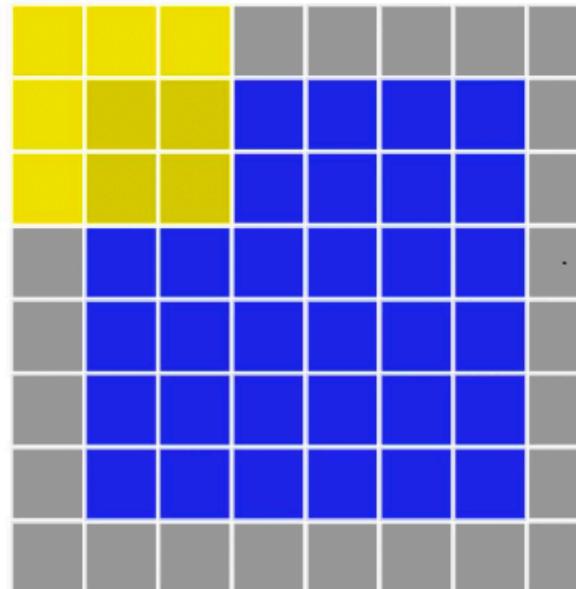
$$S = 1, F = 3,$$

$$W_{in} = H_{in} = 6$$

$$W_{out} = H_{out} = (6 - 3) / 1 + 1 = 4$$

Padding

You may have noticed that until now the convolution output is **smaller** than the input. **Padding** uses some strategy to pad the borders of our inputs to artificially increase the input size (gray). The number of pixels with which we pad is denoted P .



Example of padding with $P=1$

Padding

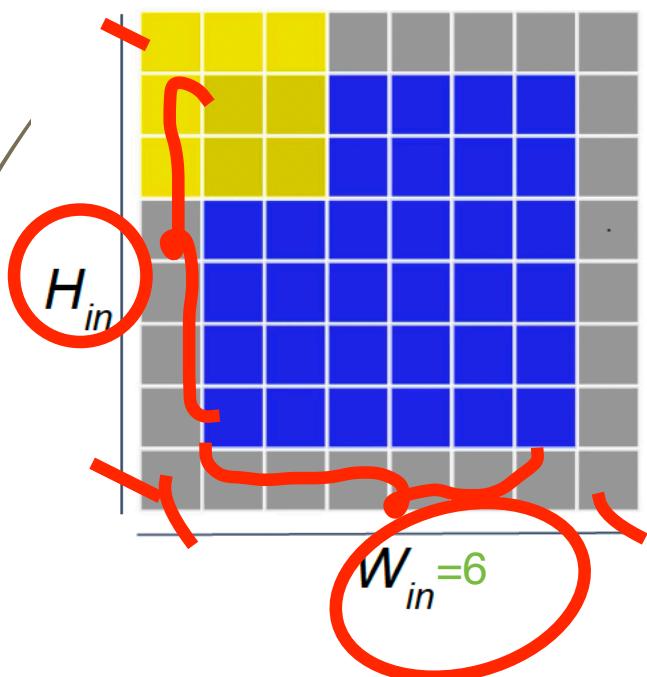
You may have noticed that until now the convolution output is **smaller** than the input. **Padding** uses some strategy to pad the borders of our inputs to artificially increase the input size (gray). The number of pixels with which we pad is denoted P .



Example of padding with $P=1$

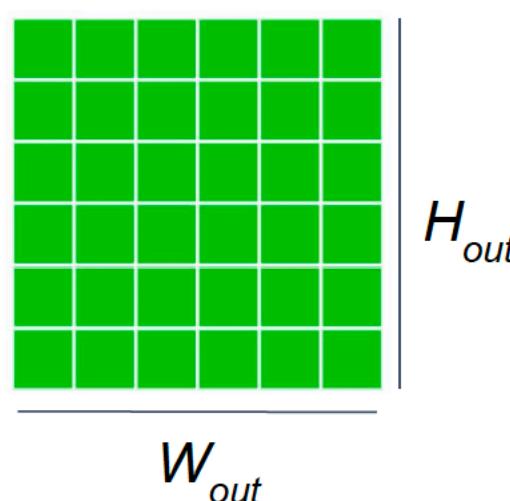
Padding

By denoting P as the number of pixels with which we pad our borders, we get:



$$W_{out} = (W_{in} - F + 2P) / S + 1$$

$$H_{out} = (H_{in} - F + 2P) / S + 1$$



In this example:

$$S = 1, D = 1, F = 3, P=1$$

$$W_{in} = H_{in} = 6$$

$$W_{out} = H_{out} = (6 - 3 + 2*1) / 1 + 1 = 6$$

Padding

In practice, there are 3 ways padding can be used to achieve different output sizes:

Valid convolution:

The kernel is allowed to visit only positions where the kernel is contained entirely within the input (i.e. no padding)

Padding

In practice, there are 3 ways padding can be used to achieve different output sizes:

Same convolution:

Same convolution
Just enough
padding is added
to keep the size of
the output equal to
the size of the input



Padding

In practice, there are 3 ways padding can be used to achieve different output sizes:

Full convolution:

Enough padding is added for every pixel to be visited exactly M times in each direction.



Padding Strategies

There are different strategies for padding. The most common one is **zero padding**, in which zeros are simply added at the borders.

```
t = tf.constant([[1, 2, 3], [4, 5, 6], [7,8,9]])
```

1	2	3
4	5	6
7	8	9

Padding Strategies

There are different strategies for padding. The most common one is **zero padding**, in which zeros are simply added at the borders.

```
t = tf.constant([[1, 2, 3], [4, 5, 6], [7,8,9]])  
paddings = tf.constant([[2, 2], [2, 2]])  
out = tf.pad(t, paddings, "CONSTANT")
```

Specify how much to pad
along each dimension

[[0	0	0	0	0	0	0]
[[0	0	0	0	0	0	0]
[[0	0	1	2	3	0	0]
[[0	0	4	5	6	0	0]
[[0	0	7	8	9	0	0]
[[0	0	0	0	0	0	0]
[[0	0	0	0	0	0	0]

Padding Strategies

There are different strategies for padding. The most common one is **zero padding**, in which zeros are simply added at the borders. Other strategies include **reflective** and symmetric padding.

```
t = tf.constant([[1, 2, 3], [4, 5, 6], [7,8,9]])  
paddings = tf.constant([[2, 2], [2, 2]])  
out = tf.pad(t, paddings, "REFLECT")
```

Specifies how much to pad along each dimension

[[9 8 7 8 9 8 7]
[6 5 4 5 6 5 4]
[3 2 1 2 3 2 1]
[6 5 4 5 6 5 4]
[9 8 7 8 9 8 7]
[6 5 4 5 6 5 4]
[3 2 1 2 3 2 1]]

Padding Strategies

There are different strategies for padding. The most common one is **zero padding**, in which zeros are simply added at the borders. Other strategies include reflective and **symmetric padding**.

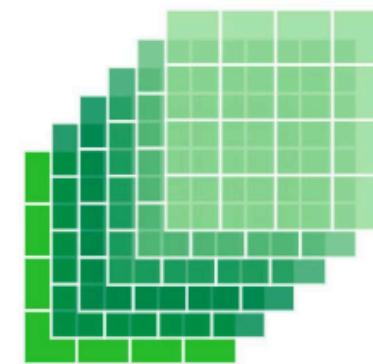
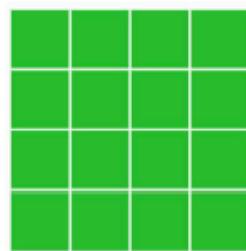
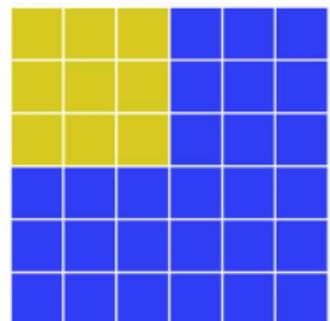
```
t = tf.constant([[1, 2, 3], [4, 5, 6], [7,8,9]])  
paddings = tf.constant([[2, 2], [2, 2]])  
out = tf.pad(t, paddings, "SYMMETRIC")
```

Specifies how much to pad along each dimension

[[5	4	4	5	6	6	5]
[2	1	1	2	3	3	2]
[2	1	1	2	3	3	2]
[5	4	4	5	6	6	5]
[8	7	7	8	9	9	8]
[8	7	7	8	9	9	8]
[5	4	4	5	6	6	5]]

Depth

We can use more than one kernel to compute many feature maps. The depth, or number of kernels, denoted as D , represents how many kernels will be used and will determine the **depth** of the volume (number of channels) of the output. The resulting feature maps are stacked to produce a volume.



Depth

We can use more than one kernel to compute many feature maps. The depth, or number of kernels, denoted as D, represents how many kernels will be used and will determine the **depth** of the volume (number of channels) of the output. The resulting feature maps are stacked to produce a volume.

Here, S=1, D=5, F=3

Bias

We add a constant bias b to every depth layer in our network. The value of the bias is learned by the network. For a depth of D , we require D biases b , one per kernel.

$$\theta = (W, b)$$

$$y = Wx + b$$

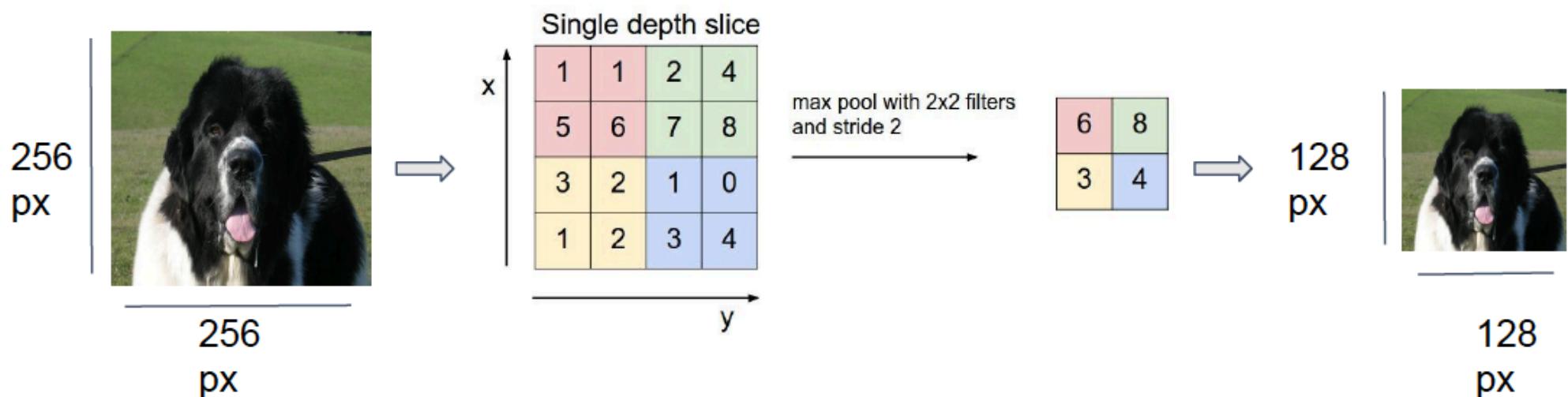
MLP: Matrix multiplication

$$y = W * x + b$$

CNN: Convolution

Pooling

Pooling is used to downsample an image. The most common type of pooling is max-pooling. Pooling makes the network less sensitive to fine details.



Pooling

Notice how we increment our kernel with a stride of 2 in both the x and y direction. This allows us to downsample by a factor of 2 in every dimension.

2x2 max-pooling with stride 2

Dilated Convolution

Another way to downsample our input is by using **dilated convolutions** (or atrous convolution). The **dilation coefficient d** determines pixel spacing between convolutions.



Dilated convolution with $d = 2$

Transposed Convolution

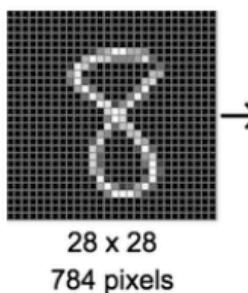
Transposed convolutions are used to increase the resolution of a feature map. It can be thought of as an interpolation, or a convolution on the output. It is often used in the context of upsampling the output of a network.

Topics

- ▶ **CNN**
- ▶ Image Representation
- ▶ Gradient Descent in CNNs

Image Representation

- ▶ An image on a screen can be defined as an array of pixels.
 - ▶ A grid of pixel intensities is referred to as a channel.
 - ▶ A grayscale image varies in white and black intensity, typically in the range [0, 255].



2D Convolution on a Grayscale Image

Image Representation

To display colours, we can use an **RGB** representation. **Red**, **Green**, and **Blue** **channels** are used to define the different **intensities** in each and combined to give all the colours our eyes can see.



https://en.wikipedia.org/wiki/RGB_color_model#/media/File:Rgb-compose-Alim_Khan.jpg

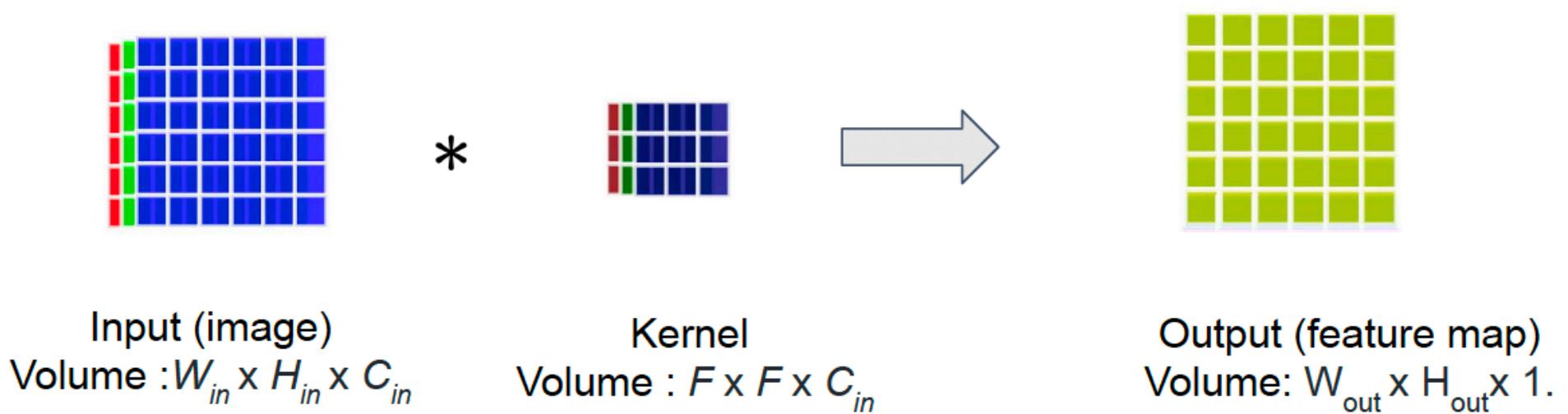
RGB Image Demo



https://en.wikipedia.org/wiki/File:Additive_colors.ogv

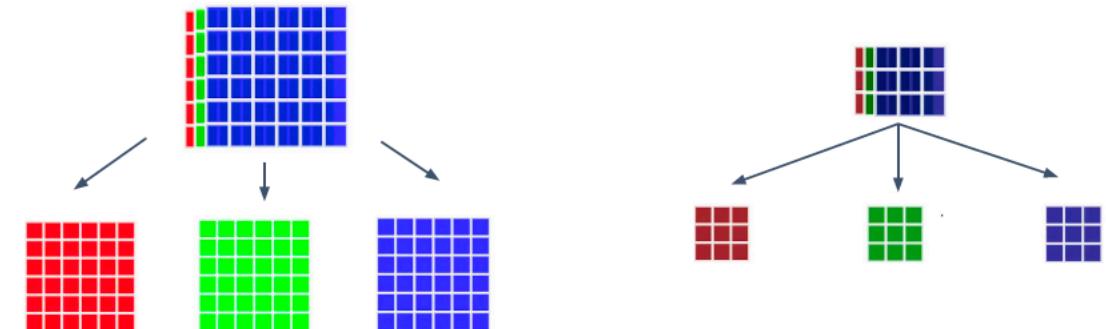
2D Convolution on an RGB Image

2D convolutions layers typically deal with volumes as **input**. Each kernel will also be a **volume**. For **each** kernel, we will have a different single-channel output.



2D Convolution

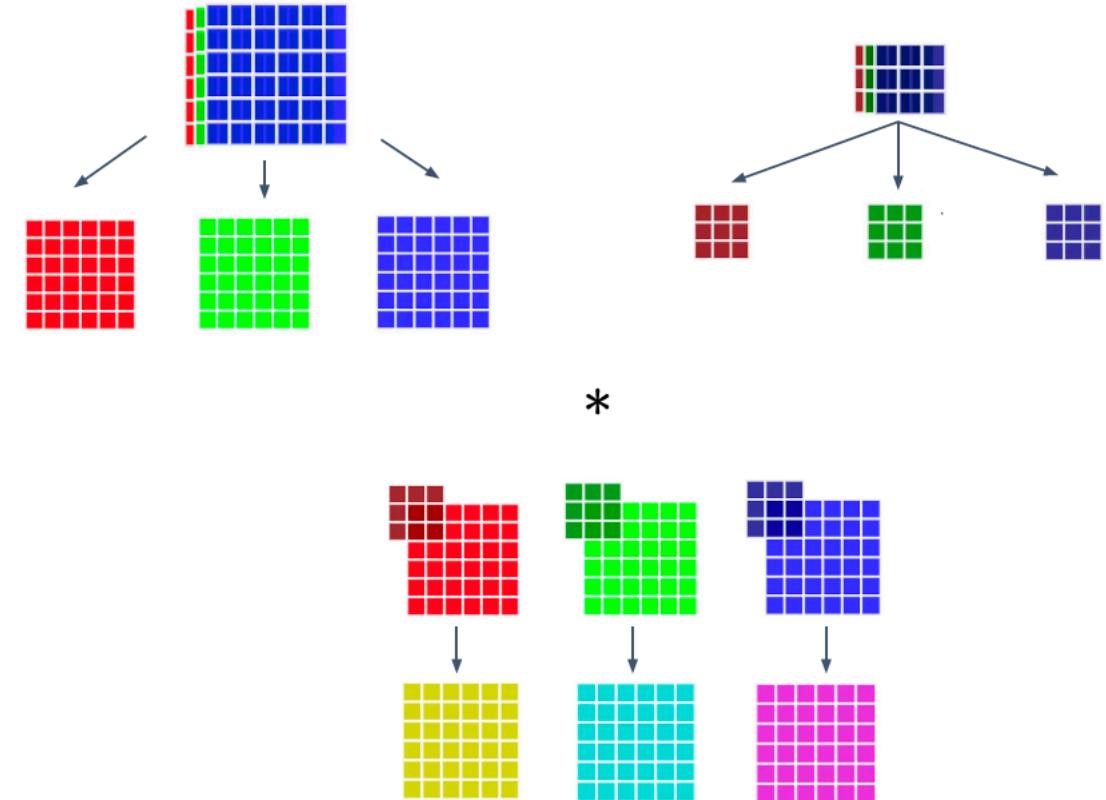
Separate the inputs and kernels by channel



2D Convolution

Separate the inputs and kernels by channel

Perform 2D convolution on each channel

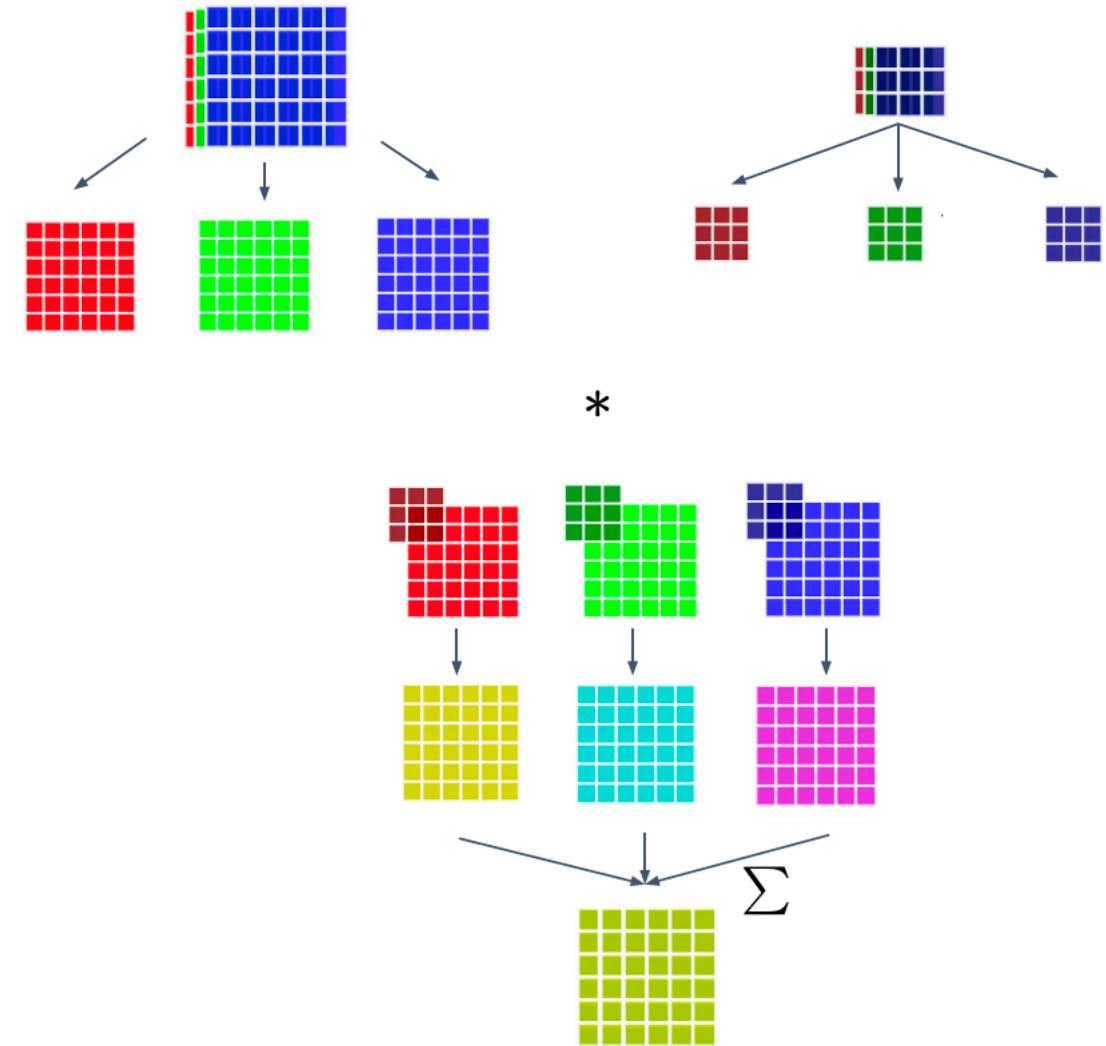


2D Convolution

Separate the inputs and kernels by channel

Perform 2D convolution on each channel

Sum the results to get a
 $W_{out} \times H_{out} \times 1$ feature map



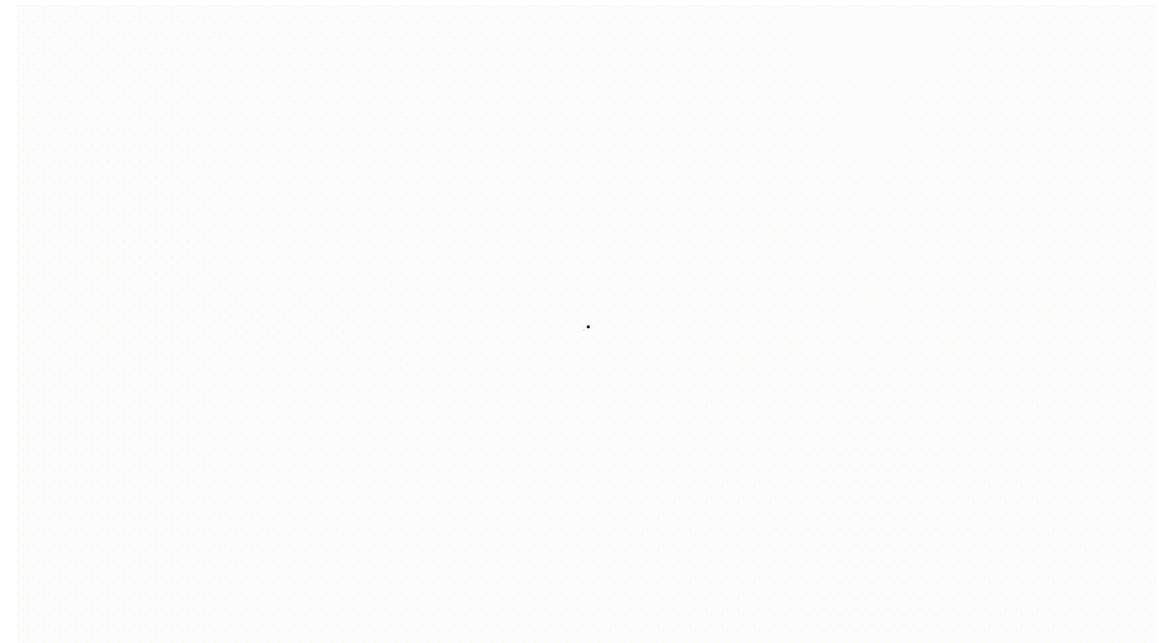
2D Convolution on an RGB Image

Here is an animation showing the entire procedure of 2D convolution using a single kernel:

Input (image)

Kernel

Feature Map (output)



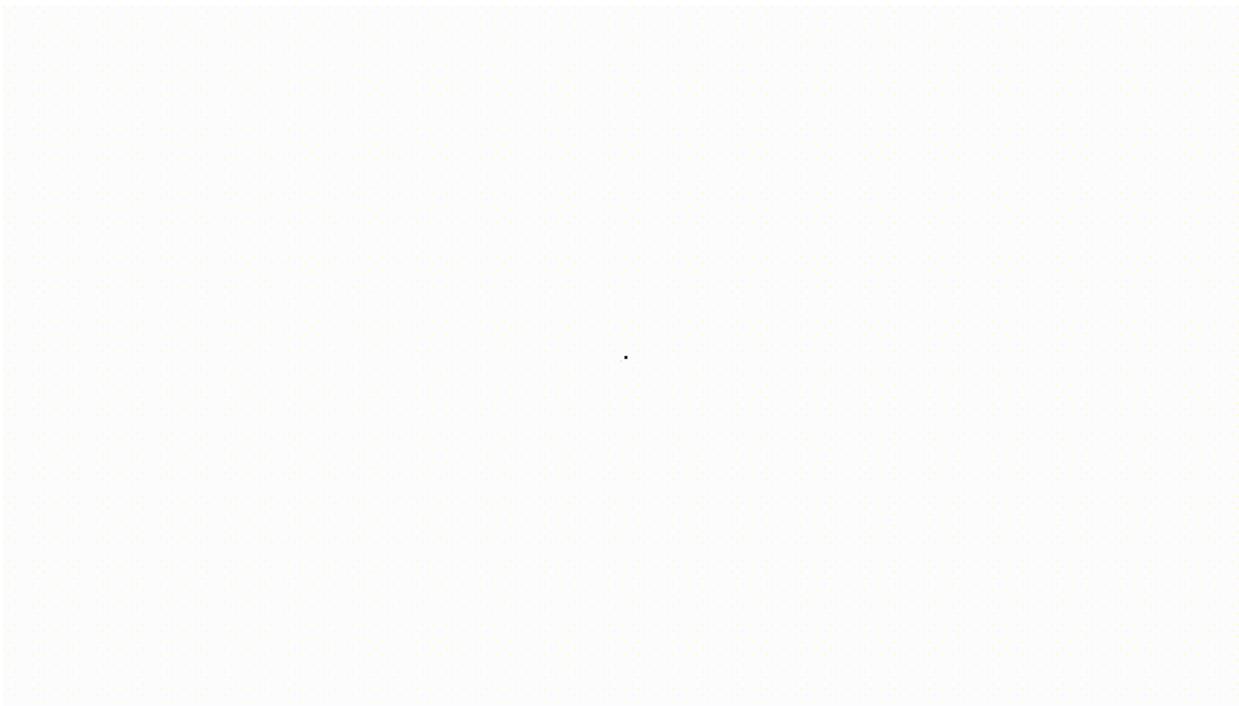
From Volume to Volume

To create depth, we simply use **multiple kernels**. Here, we are using $D=4$ kernels, each of volume $3 \times 3 \times 3$. We are going from an $6 \times 6 \times 3$ RGB image to an $6 \times 6 \times 4$ feature map. The number of kernels determine the depth, D , or C_{out} .

Input (RGB image),
 $W_{in} \times H_{in} \times 3$
 $(6 \times 6 \times 3)$

Kernels
 $D \times F \times F \times C_{in} =$
 $(4 \times 3 \times 3 \times 3)$

Feature Map
 $W_{out} \times H_{out} \times D =$
 $(6 \times 6 \times 4)$



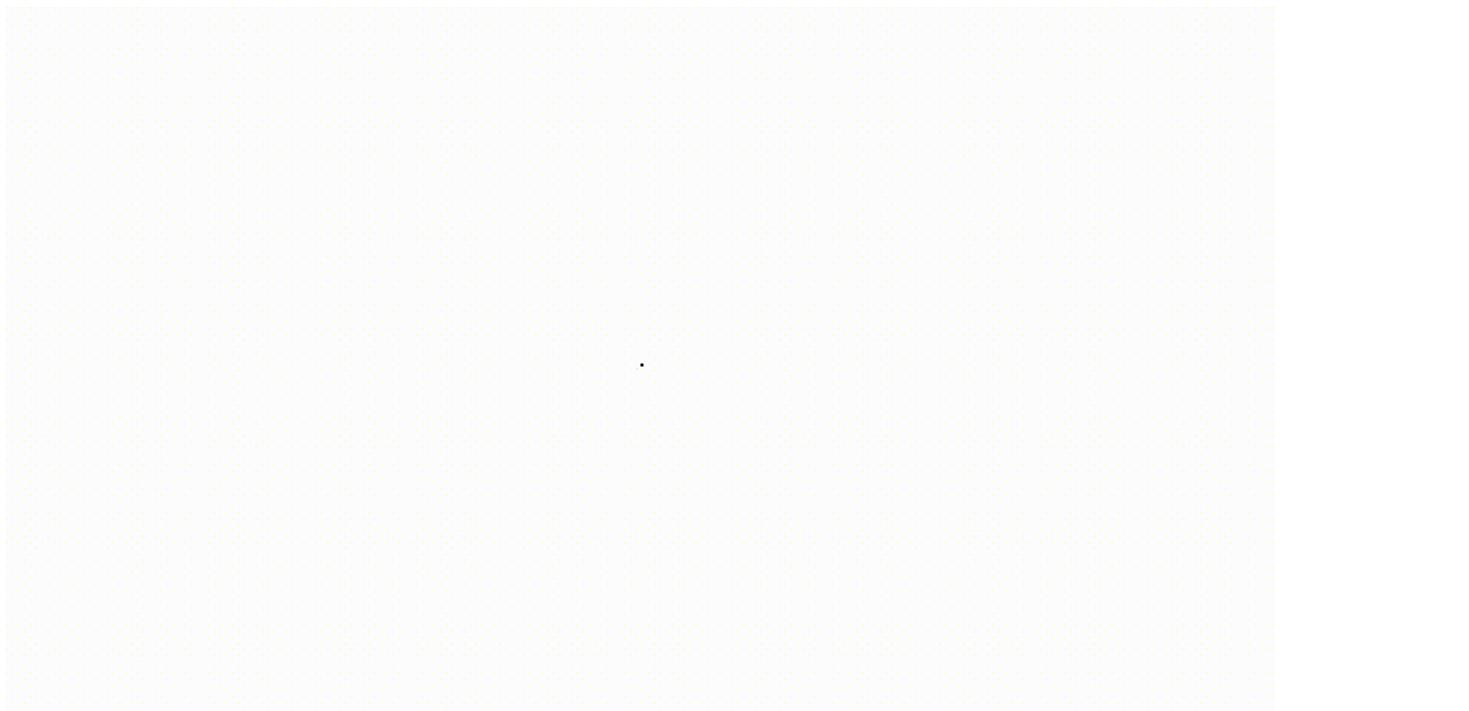
From Volume to Volume

To create depth, we simply use **multiple kernels**. Here, we are using $D=8$ kernels, each of volume $3 \times 3 \times 4$. We are going from an $6 \times 6 \times 4$ feature map to an $6 \times 6 \times 8$ feature map. The number of kernels determine the depth, D , or C_{out} .

Input (RGB image),
 $W_{in} \times H_{in} \times 4$
 $(6 \times 6 \times 4)$

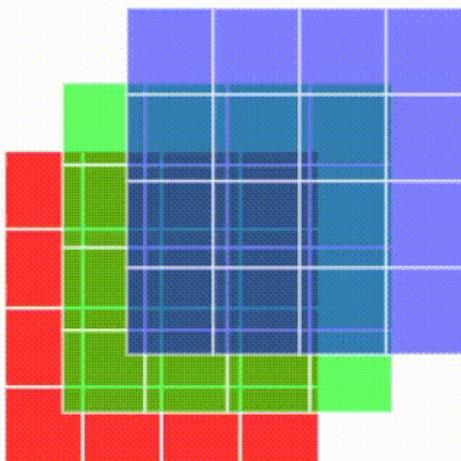
Kernels
 $D \times F \times F \times C_{in} =$
 $(8 \times 3 \times 3 \times 4)$

Feature Map
 $W_{out} \times H_{out} \times D =$
 $(6 \times 6 \times 8)$



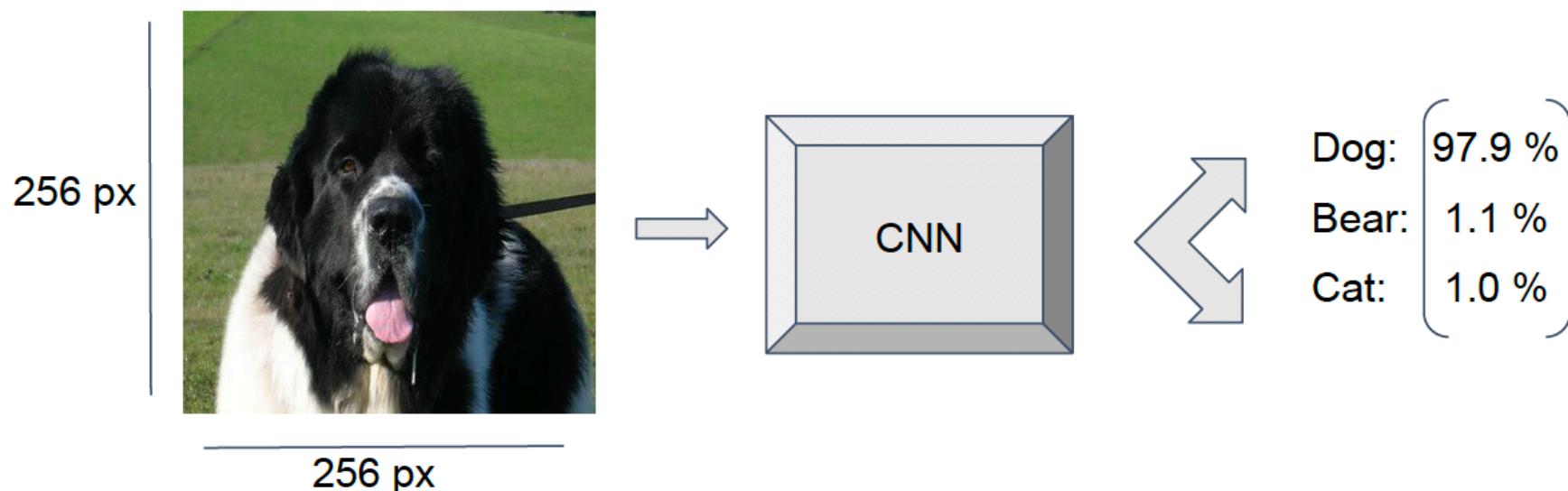
1x1 Convolutions

1x1 convolutions are useful in the context of non-linear channel reduction. They allow us to preserve the width and height and reduce on the depth. They also require few parameters.



CNN Building Blocks

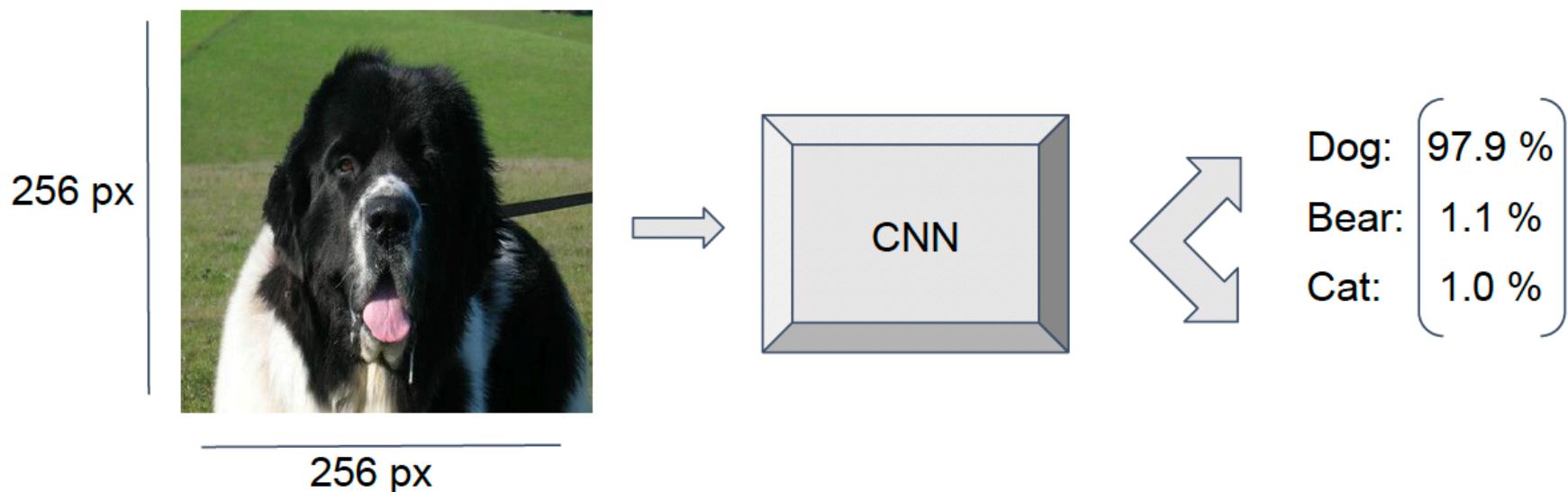
A CNN can be used to map an input (i.e. an image) to a **probability distribution**.



Building Blocks

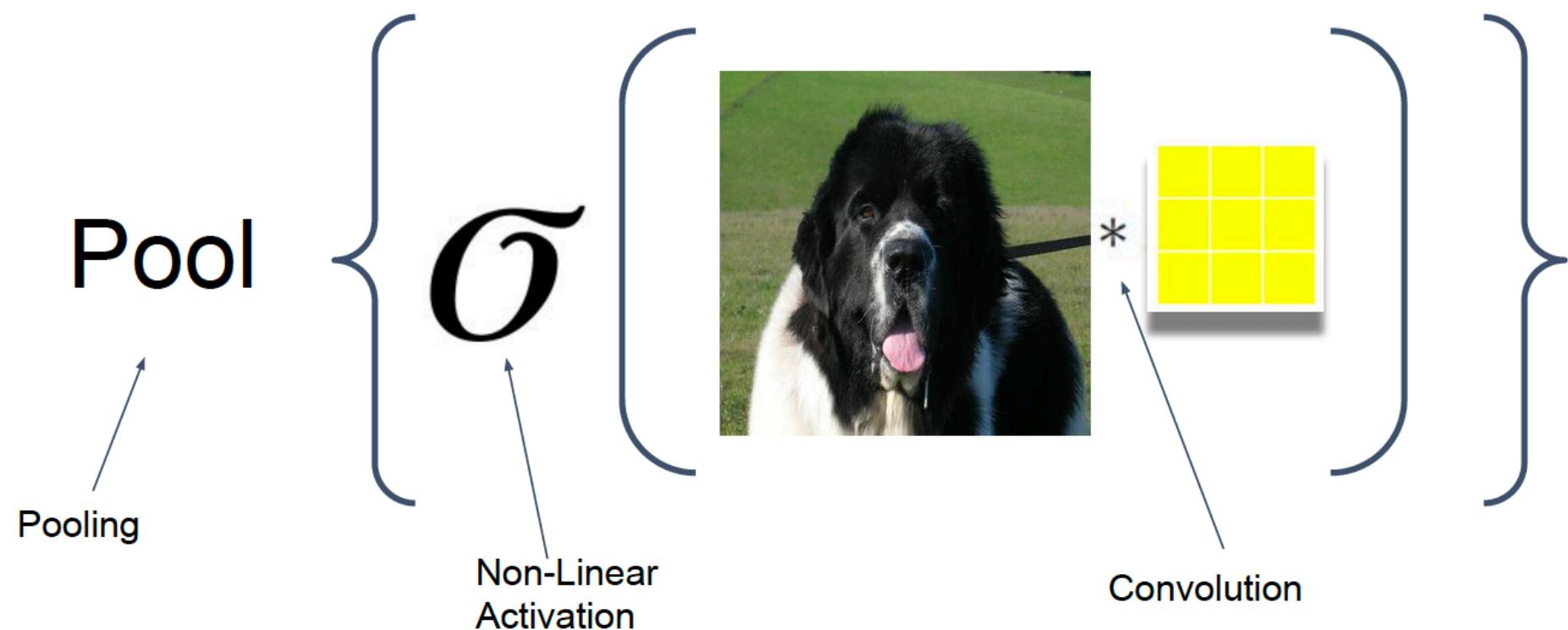
Consider a colour image of 256x256 pixels. We have a total of:
256 pixels *256 pixels *3 channels = 196 608 inputs.

We want to reduce it to a vector of size 1xM, containing the probability of belonging to each of the M classes.



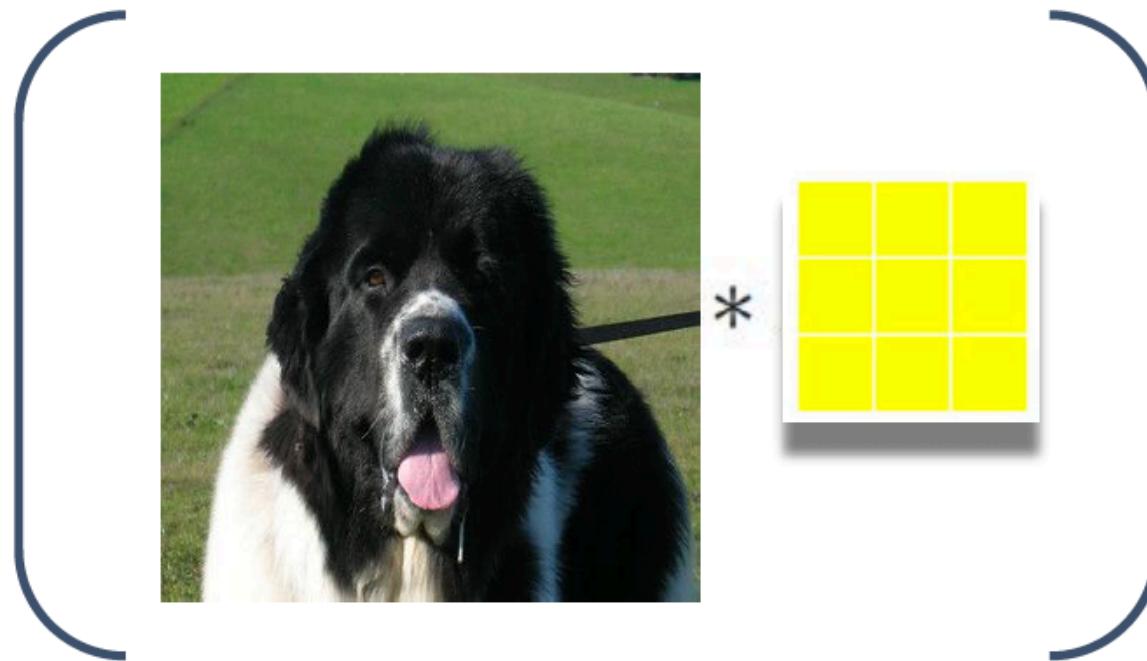
Building Blocks

We typically use a series of building blocks. A common building block for a CNN is shown here:



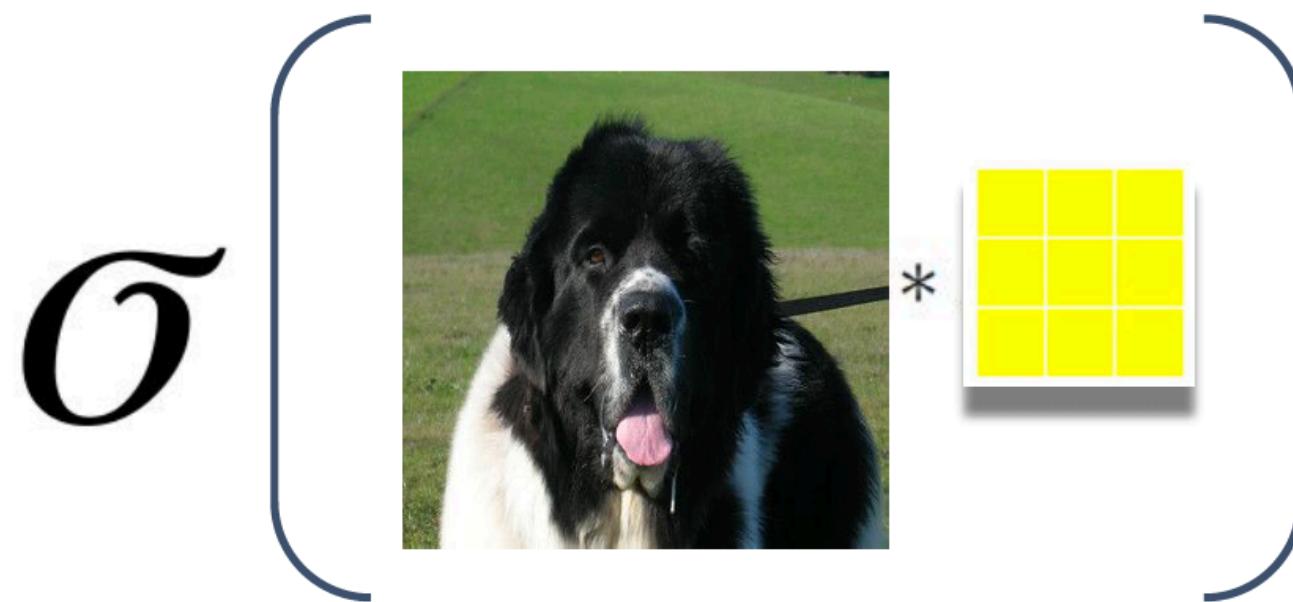
Building Blocks

Step 1: **Convolution** - Apply the convolution operation to our input with learned kernels



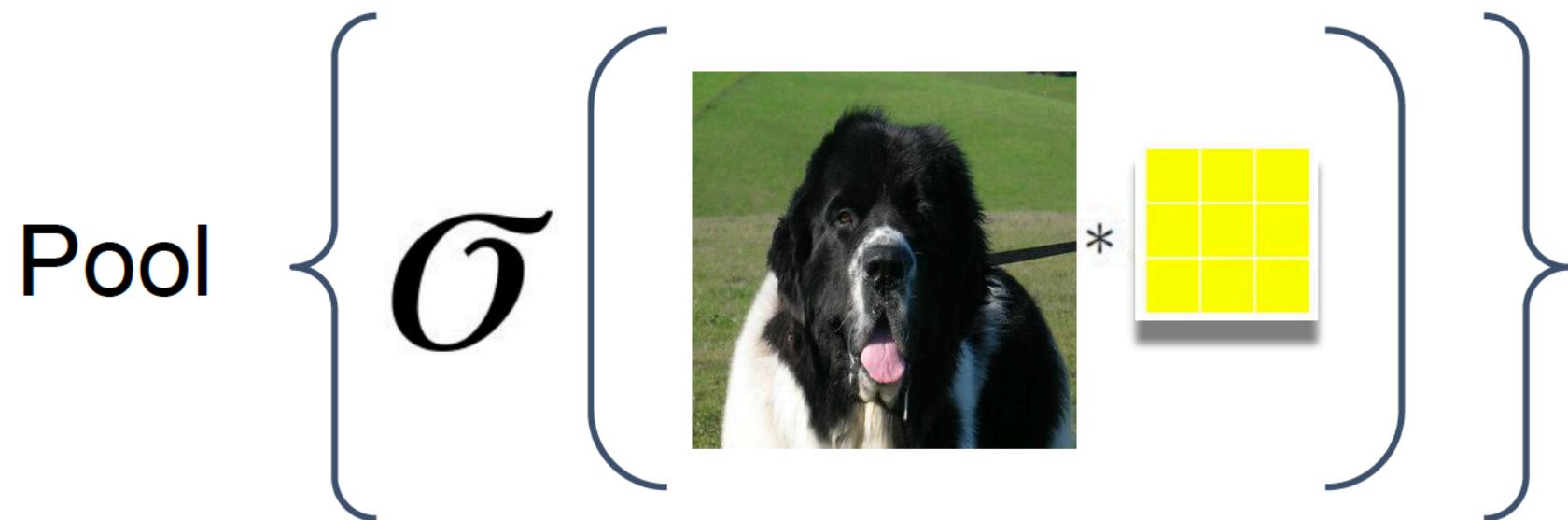
Building Blocks

Step 2 : **Activation** - Apply a non-linear activation function element-wise to the output of the convolution. This can be i.e. a $\text{ReLU}(x)$ or $\tanh(x)$.



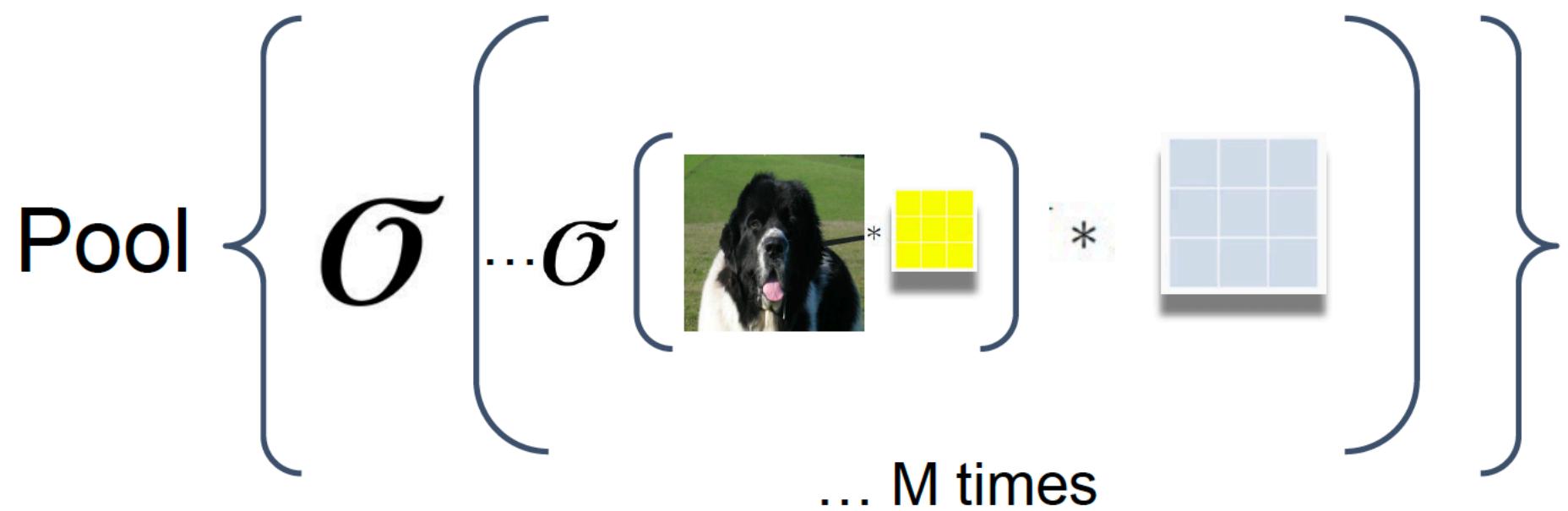
Building Blocks

Step 3: **Pooling** - Apply the pooling operation to the output of the activation function. Max Pooling is commonly used in modern architectures.



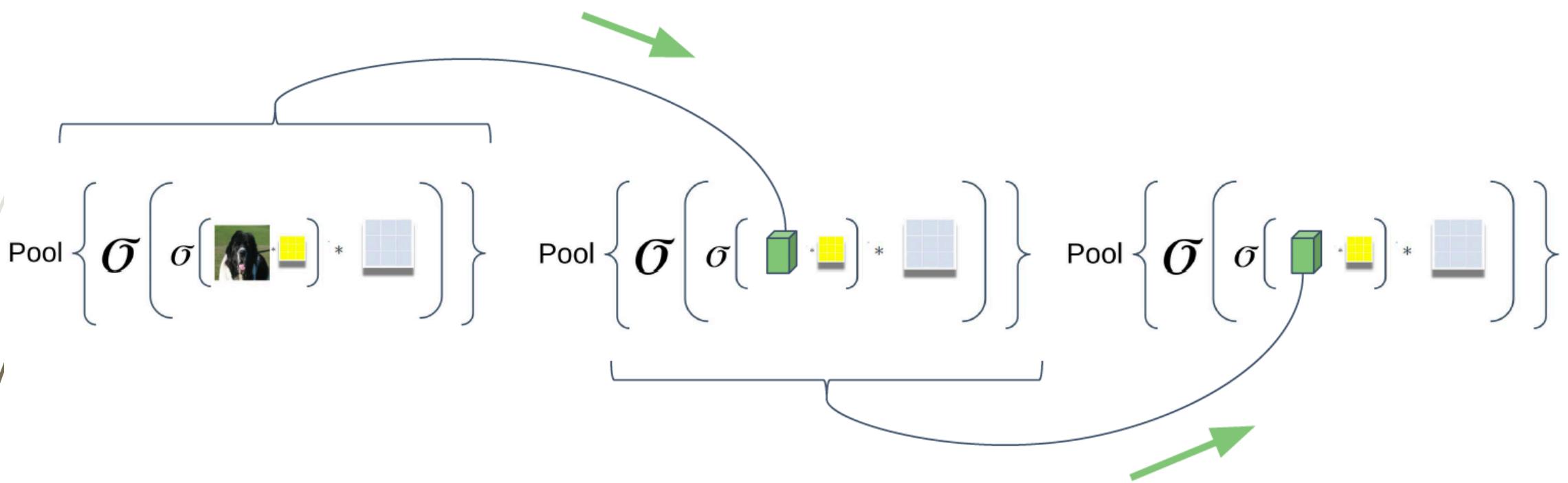
Building Blocks

We can have many *Convolutions + Activations* chained before pooling:



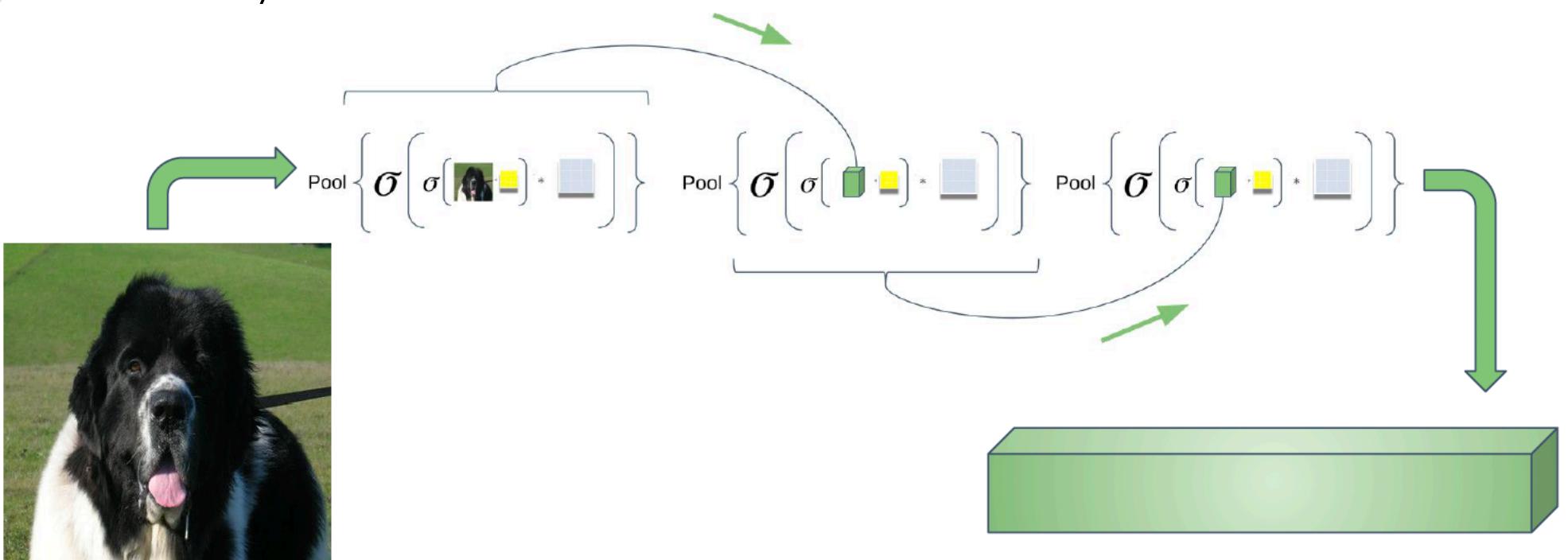
Building Blocks

... And we can have many of these blocks chained together



Building Blocks

These blocks allow us to map from one volume to another of arbitrary dimensions



$$V_{in} = W_{in} \times H_{in} \times D_{in}$$

$$V_{out} = W_{out} \times H_{out} \times D_{out}$$

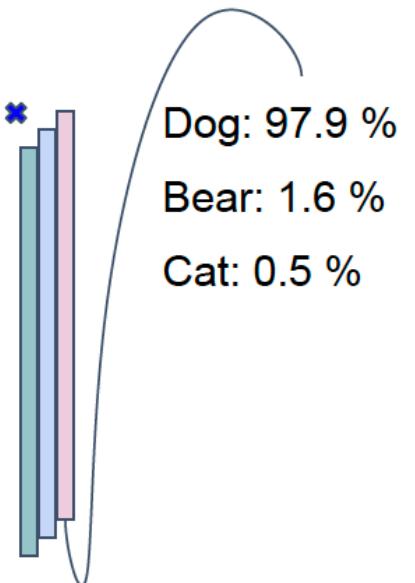
Building Blocks

To get to a probability distribution, we need to obtain a 1D vector of dimension 1xM (M categories). One approach is to **flatten** out the feature map volume and then use a dense **fully connected layer** followed by a **softmax** to obtain a **probability distribution** that sums to 1.



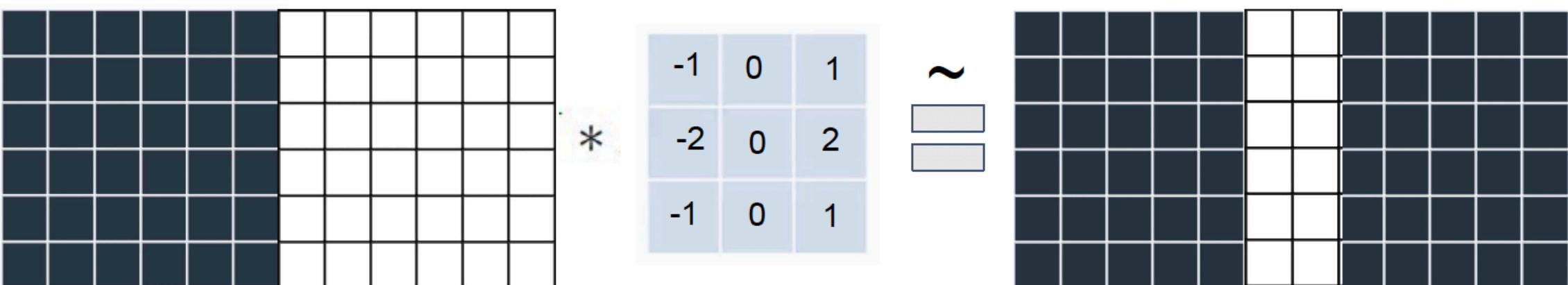
$$V_{out} = W_{out} \times H_{out} \times D_{out}$$

$$V_{in} = W_{in} \times H_{in} \times D_{in}$$



Motivation for CNNs

- ▶ CNNs can learn useful features from images automatically
- ▶ Classically, these features were handcrafted. For example, a Sobel filter was designed to extract edges. Only at the border will the output of the convolution be non-zero.

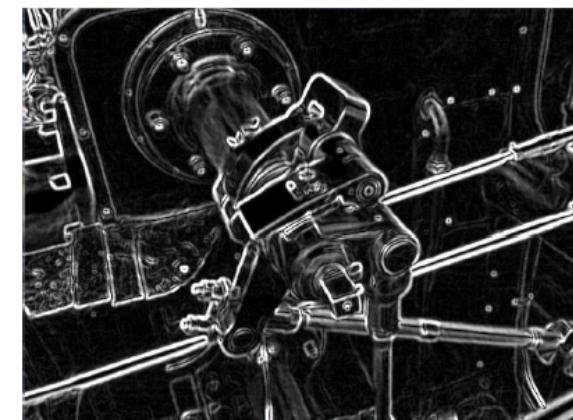


Motivation for Convolutions

We can combine a horizontal and vertical Sobel filter to extract edges from the image.

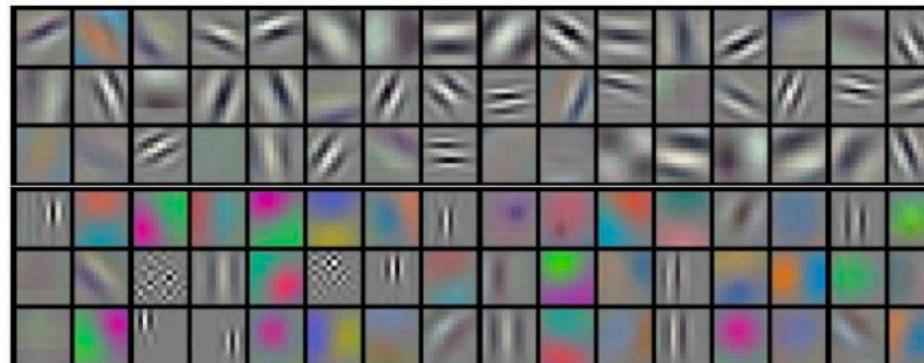


$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} * \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$



Motivation for Convolutions

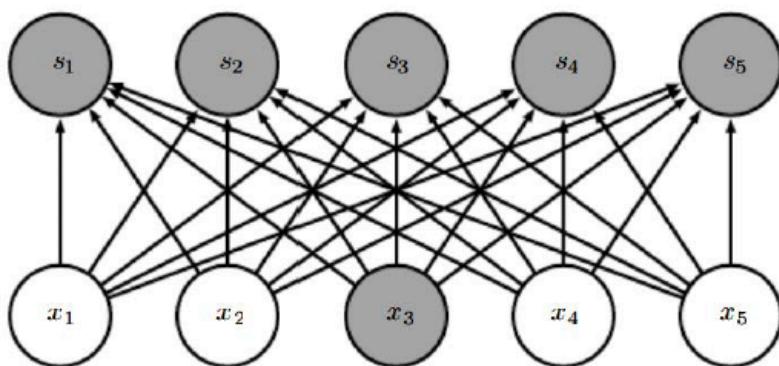
- ▶ It can be very hard and time consuming to handcraft these filters. Instead, we use deep learning to learn the best filters.
- ▶ By applying convolutions to images, we can extract some high-level features describing the images but it can be very challenging and limiting.
- ▶ By applying convolutions to features, we can extract features of features at different levels of abstraction.



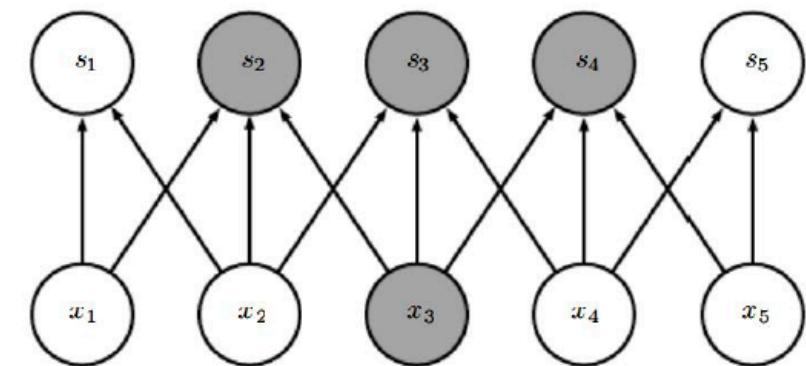
Filters learned by AlexNet on ImageNet.

Motivation for Convolutions

In MLPs, each input can in theory contribute equally to a signal. CNNs introduce a way to guide the neural network in focusing **hierarchically** at local features of an image.



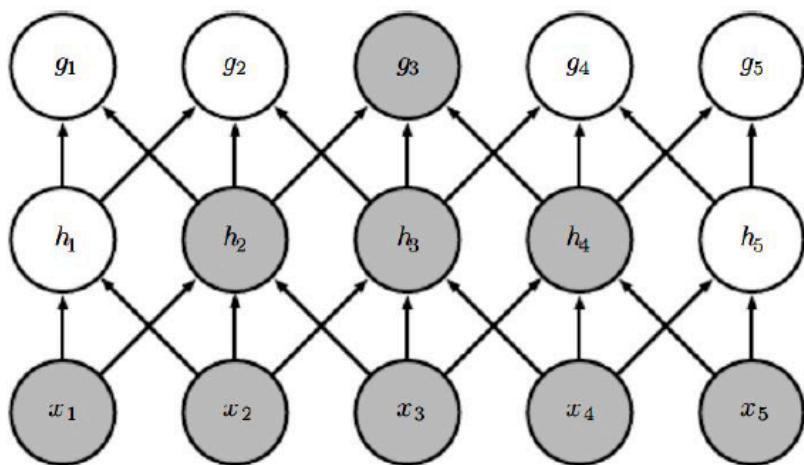
MLP



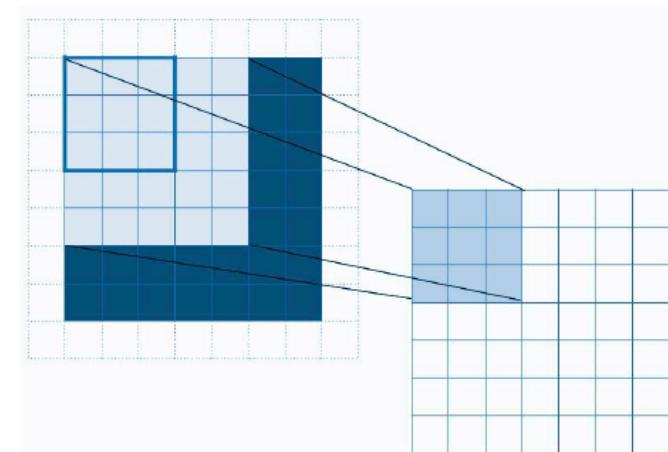
CNN

Motivation for Convolutions

Receptive field - Image patches contain information from larger patches of previous inputs. The resulting 3x3 patch seen here contains information from the previous 5x5 patch. The deeper you go, the wider the receptive field.



Field of view for 1D system



Field of view for 2D system

Motivation for Convolutions

► Parameter sharing

- weights are used at different areas of a same image which requires significantly less memory for models.
- kernels can extract similar features at multiple places in an image.
- Convolutions are **equivariant to translation** - i.e. translating an input in space will result in a translated convolution output.
- Useful features which have been learned can be **reused** at multiple regions in an image.

Hello (CNN) World

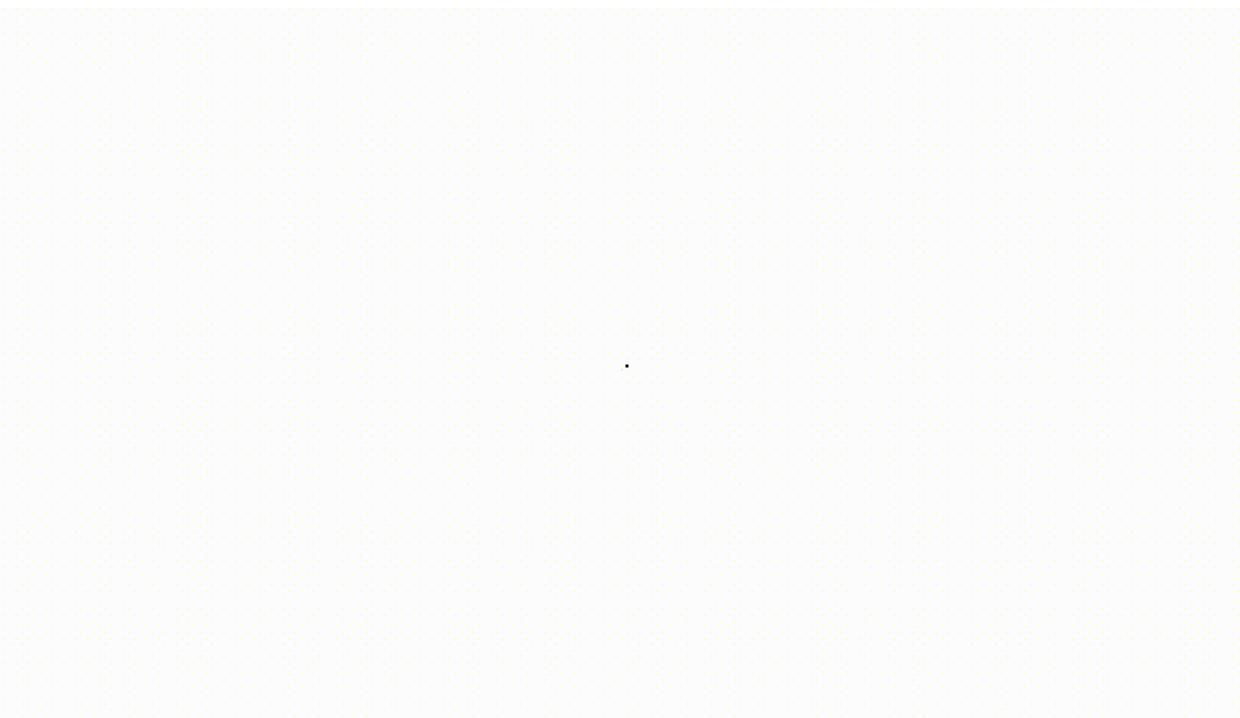
Let's now consider a real example of a CNN that has been around for a long time: **LeNet**. It was designed by Yann Lecun et al. to classify handwritten digits (0-9) from the **MNIST** dataset.

0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
4	4	4	4	4	4	4
5	5	5	5	5	5	5
6	6	6	6	6	6	6
7	7	7	7	7	7	7
8	8	8	8	8	8	8
9	9	9	9	9	9	9

The MNIST dataset contains approximately 60 000 handwritten digits with their associated label.

MNIST and LeNet

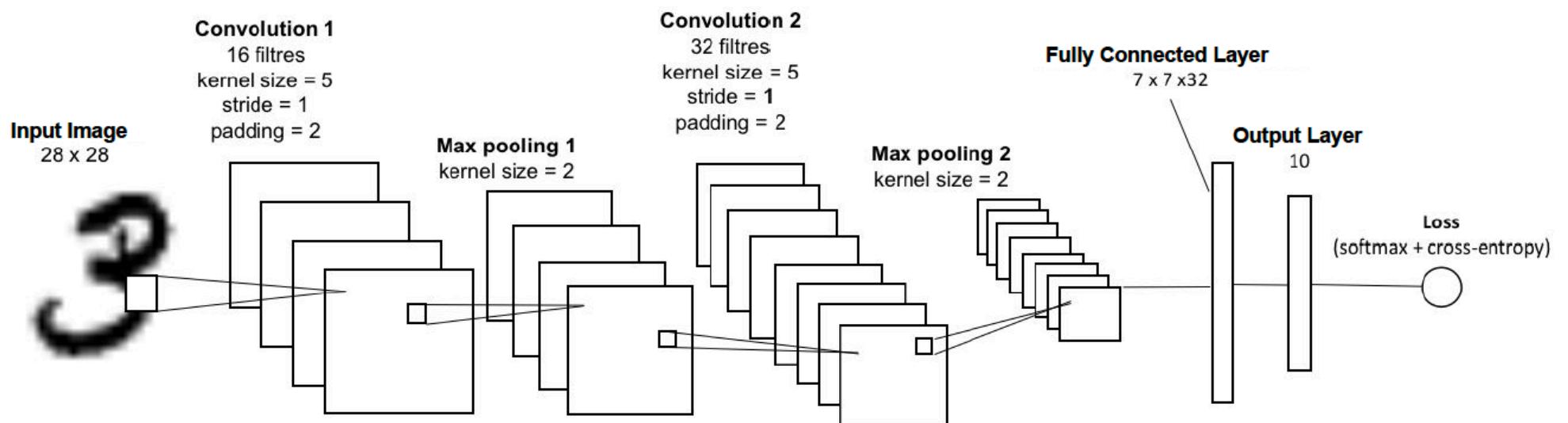
Problem statement : Given an **input** grayscale image, we want to output the **probability** of it being a certain digit.



0:	1.6 %
1:	1.2 %
2:	0.1 %
3:	1.25%
4:	2%
5:	1.3%
6:	1.2 %
7:	0.3%
8:	90.95%
9:	0.1%

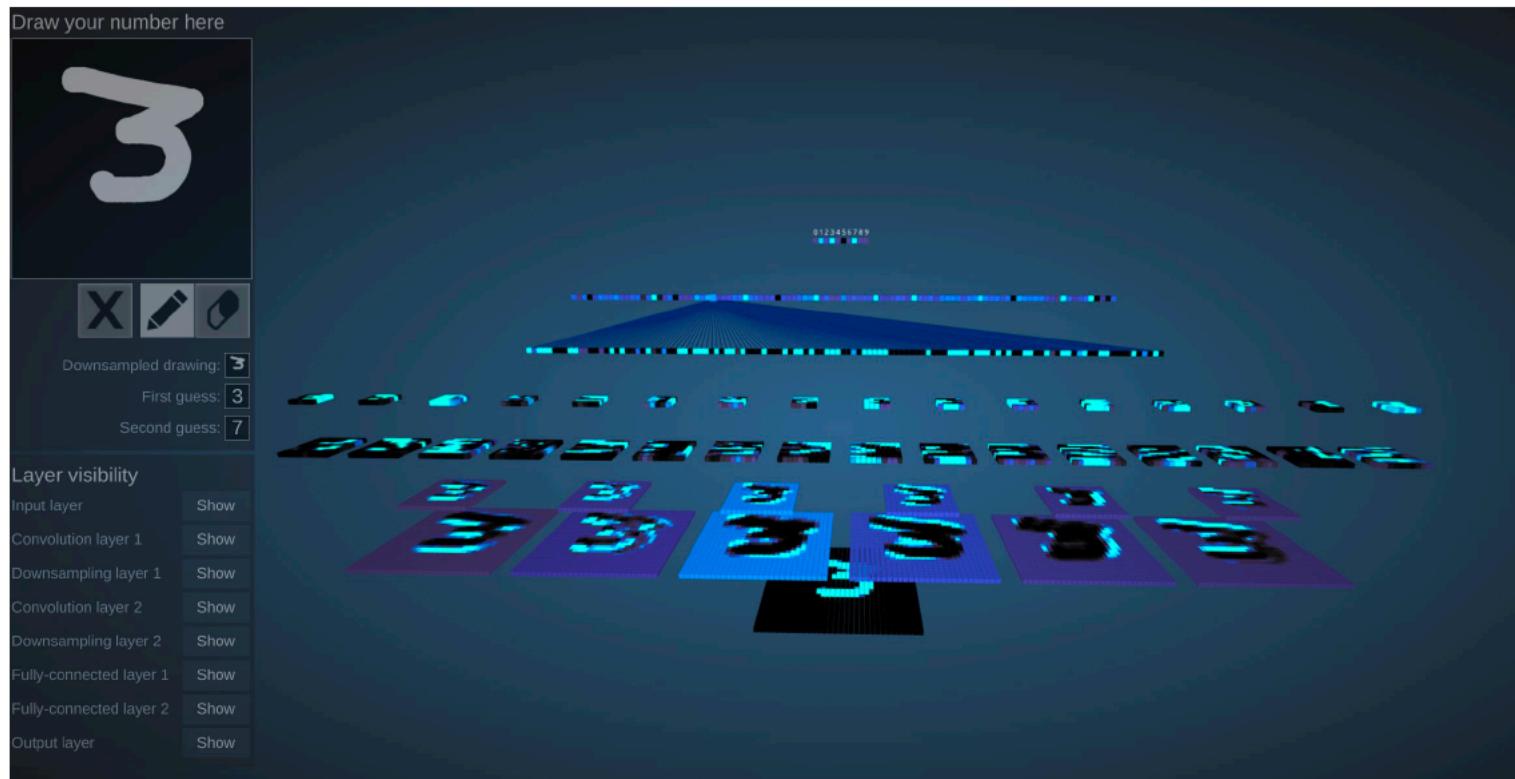
LeNet

LeNet is considered a “simple” architecture. The network is comprised of convolutions, pooling and fully connected layers.



Demo

Here is a cool link to visualize what is happening in LeNet:
<http://scs.ryerson.ca/~aharley/vis/conv/>



The End

