4 - Convolutional Neural Networks

Marco Willi

Motivation

CNNs work similarly to MLPs: They consist of neurons with weights and biases arranged in layers. CNNs also have an output with which a differentiable loss function can be calculated so that the weights and biases can be adjusted using backpropagation.

Unlike MLPs, CNNs explicitly assume that inputs (e.g., pixels) that are close together need to be considered together and that information is locally correlated. This allows certain properties to be embedded in the architecture of CNNs (inductive biases) to define models much more efficiently (with fewer parameters).

The input to an MLP is a vector $\mathbf{x}^{(i)}$, which is transformed through multiple hidden layers to the output layer. Each hidden layer has a certain number of neurons, each connected to all neurons in the previous layer (fully-connected layers), see Figure 1.

The fully connected layers can only process 1-D vectors. Therefore, images $\in \mathbb{R}^{H \times W \times C}$ must be flattened into 1-D vectors $\in \mathbb{R}^{p}$. Here, $p = H \times W \times C$. This causes MLPs to become very large (having many learnable parameters) when applied to high-dimensional inputs such as images. In the CIFAR-10 dataset, which consists of very small images of 32x32x3 (height, width, colors), a single neuron in the first hidden layer has 32 * 32 * 3 = 3,072 weights to learn (see Figure 3 for an illustration in an MLP and Figure 2 for an illustration on a linear model).

For larger images, which are often encountered in practice, the number of weights is correspondingly much larger. Many neurons are also used, further increasing the number of parameters, leading to overfitting, and making learning the weights more difficult.

A single neuron in a CNN is only connected to a small portion (local connectivity) of the image (see Figure 4). As a result, the neurons have far fewer parameters than in an MLP. The 2-D structure of the image is also preserved, meaning they do not need to be flattened as in an MLP. This exploits the property of images that certain features, such as edges and corners, are relevant throughout the image. By convolving the neurons across the entire input, the same feature can be detected by a neuron throughout the image. In an MLP, a specific feature would need to be relearned at each position.

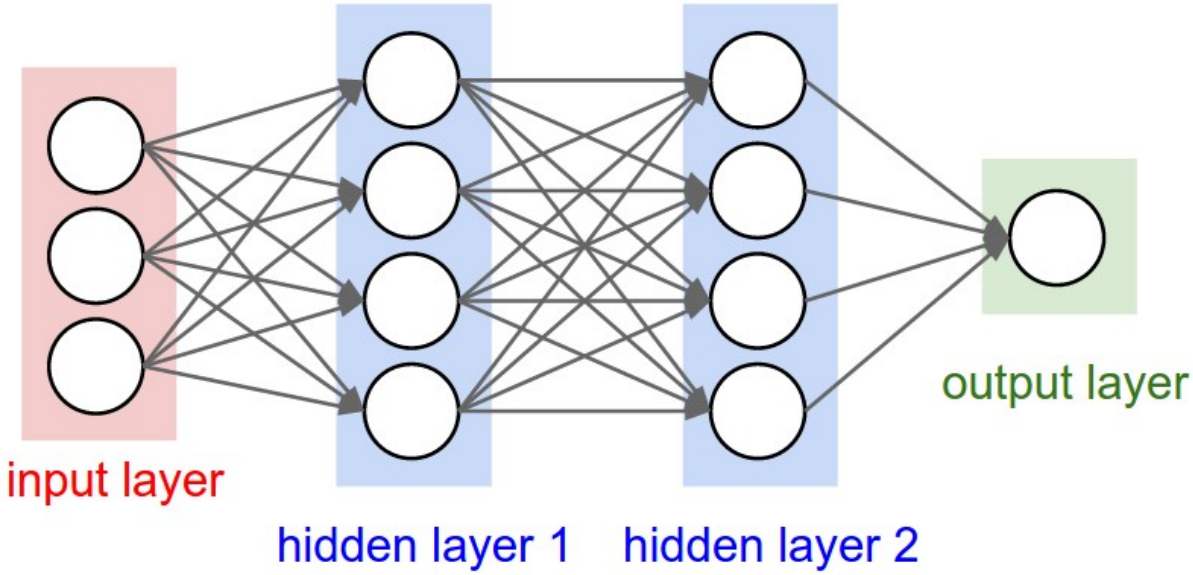


Figure 1: Source: Li (2022)

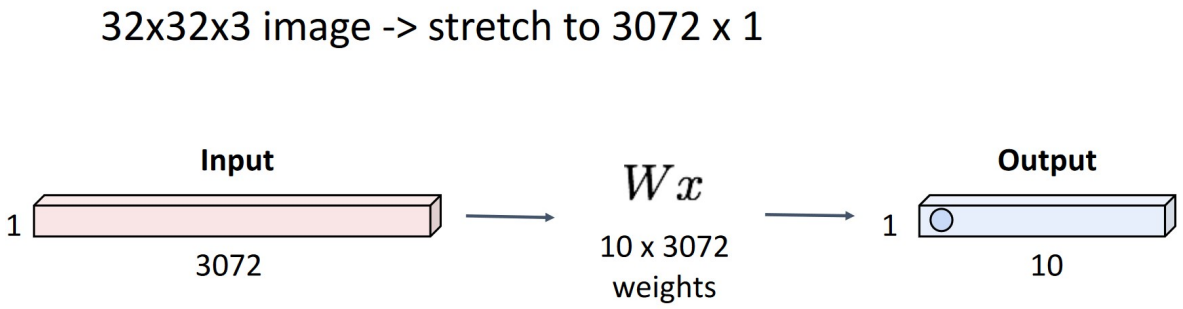


Figure 2: Source: Johnson (2019)

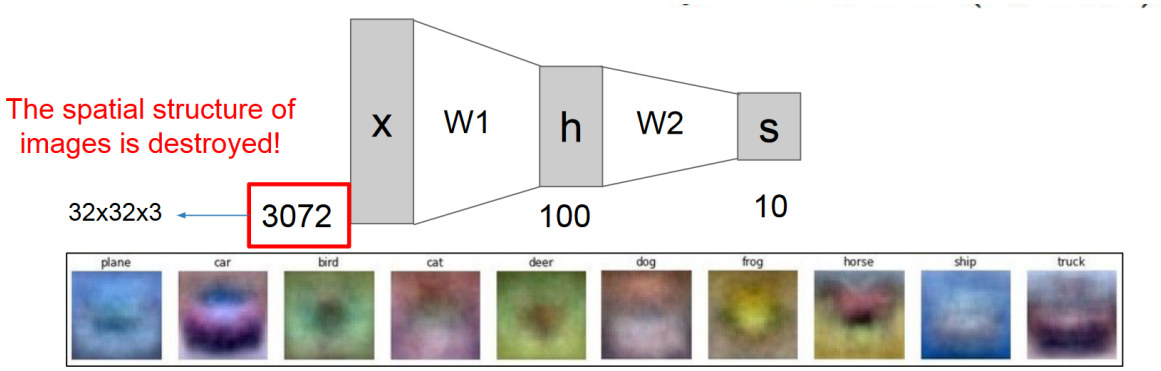


Figure 3: Source: Li (2023)

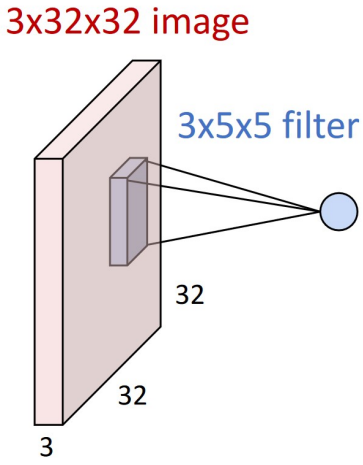


Figure 4: Source: Johnson (2019)

> **Note**
>
> CNNs are used not only for image data but also for data with spatial dependencies/local structures. This includes not only images but also time series, videos, audio, and text. The key is that signals that are spatially close together should be interpreted together.

Architecture

CNNs consist of a sequence of different layers. Each layer transforms activations from the previous layer into new activations through a differentiable operation. Below we look at the main layer types: convolutional layers, pooling layers, activation layers, and fully connected layers. Arranged in a specific sequence, this is referred to as the architecture of the model.

Figure 5 shows an example architecture. The activation maps of the various layers are shown, representing the corresponding outputs of the layers.

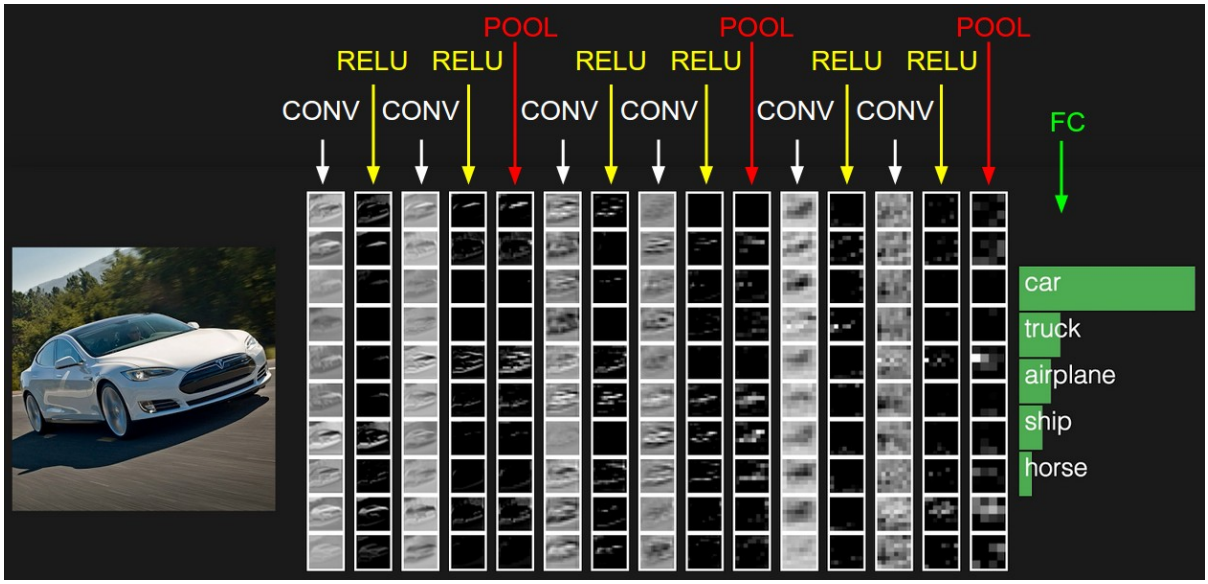


Figure 5: The activations of a ConvNet architecture are shown. The input image is on the left and the predictions on the right. Source: Li (2022).

Sometimes different layers are combined and referred to as a block. For example, the combination of a convolutional layer followed by an activation layer and a pooling layer is often used. This would be a CONV-ACT-POOL block.

Convolutional Layers

Convolutional layers are the main layers in CNNs responsible for extracting visual features. The weights of a convolutional layer consist of a set of learnable filters. Each filter is typically small along the spatial dimensions (height, width) relative to the input but extends over the entire input depth. A typical filter in the first layer, for example, has the dimension $7 \times 7 \times 3$ (7 pixels along height/width and 3 along the color channels). During the forward pass, the filters are convolved along height/width over the input. At each position, the dot product (when considering the input and filter as 1-D vectors) between the filter and input is calculated. This produces a 2-D activation map representing the filter's expression at each position in the input. Intuitively, the CNN learns filters corresponding to typical visual patterns, such as edges and colors. A set of K filters produces activation maps with a depth of K.

> **Note**
>
> Filter and kernel are sometimes used synonymously. Here, we differentiate by considering a filter as 3-dimensional (CxHxW) and a kernel as 2-dimensional (HxW). A filter consists of C kernels.

> **Note**
>
> Convolution in deep learning is typically implemented as cross-correlation.
> Given:
>
> - Kernel K
> - Image I
>
> $$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{1}$$

The data is processed in mini-batches, i.e., multiple images at once, as shown in Figure 10.

Hyper-Parameters

To define a convolutional layer, various hyperparameters need to be set. Some of the most important ones are:

- Depth
- Padding
- Stride
- Kernel Size

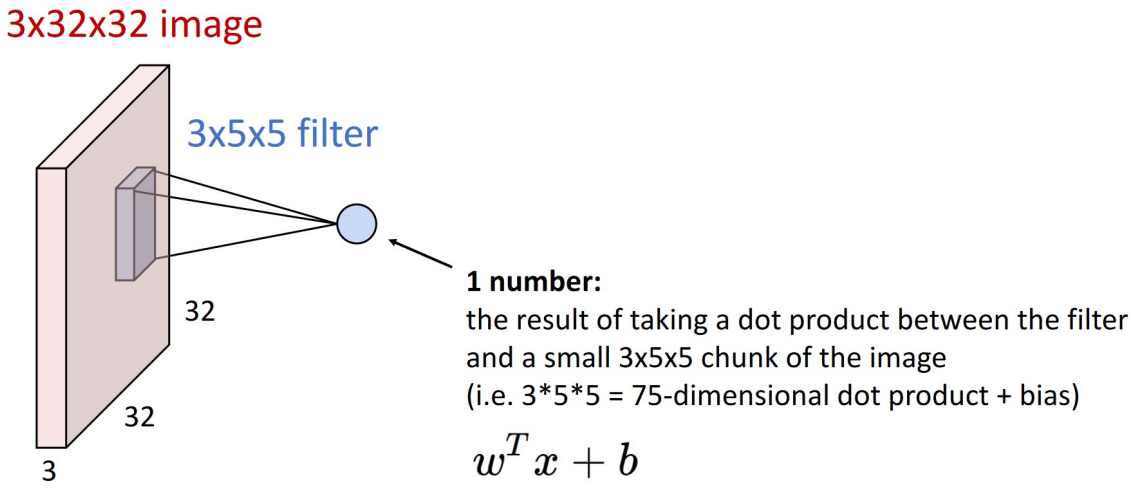


Figure 6: Source: Johnson (2019)

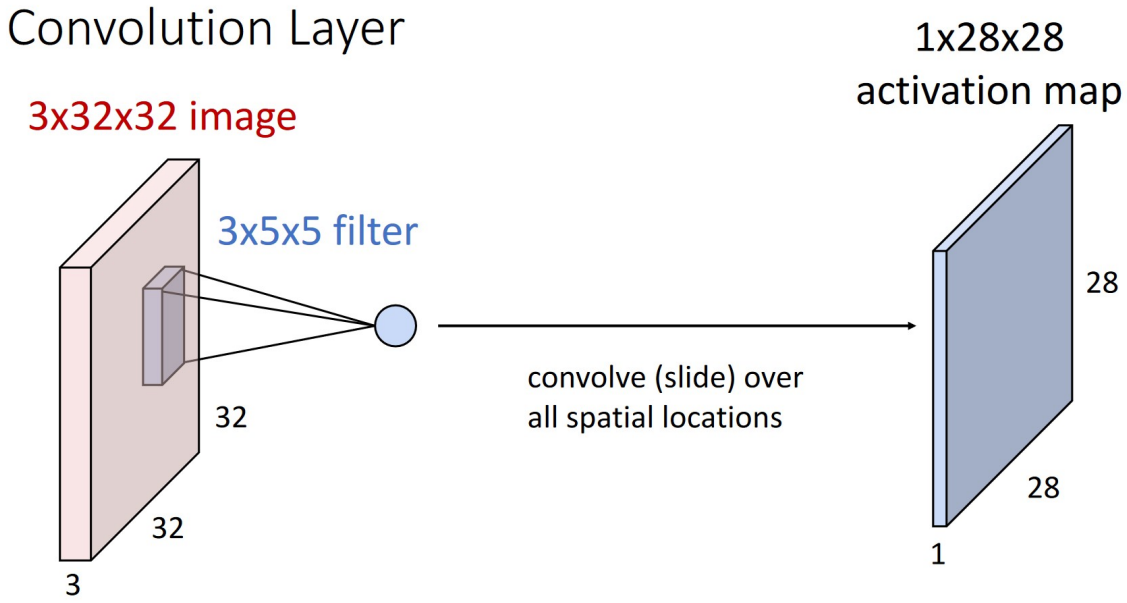


Figure 7: Source: Johnson (2019)

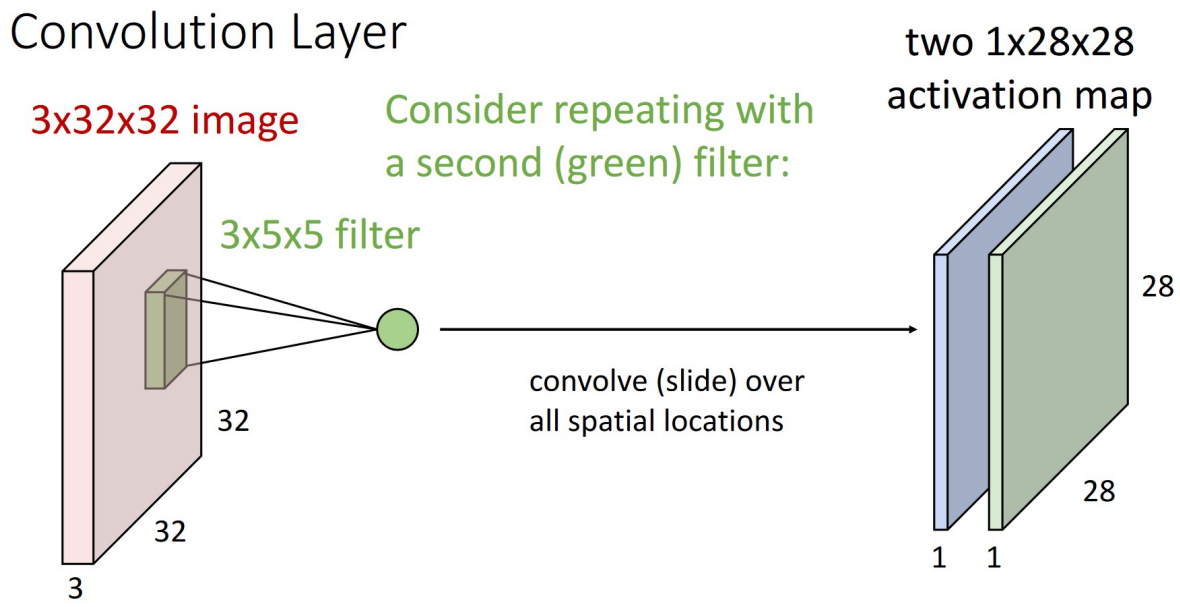


Figure 8: Source: Johnson (2019)

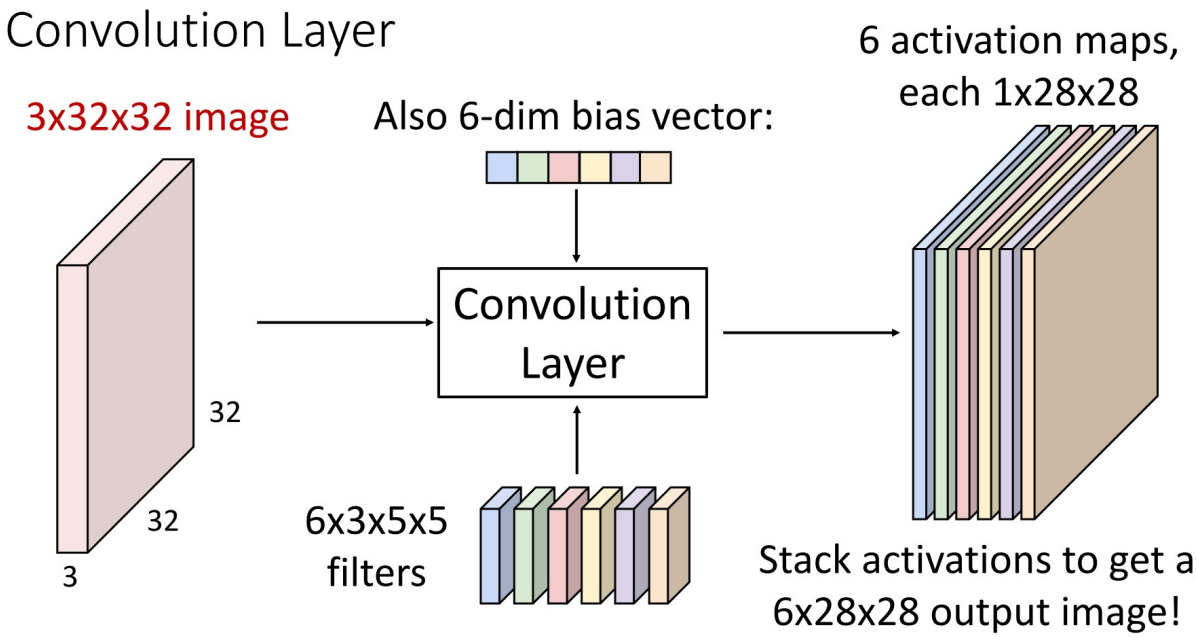


Figure 9: Source: Johnson (2019)

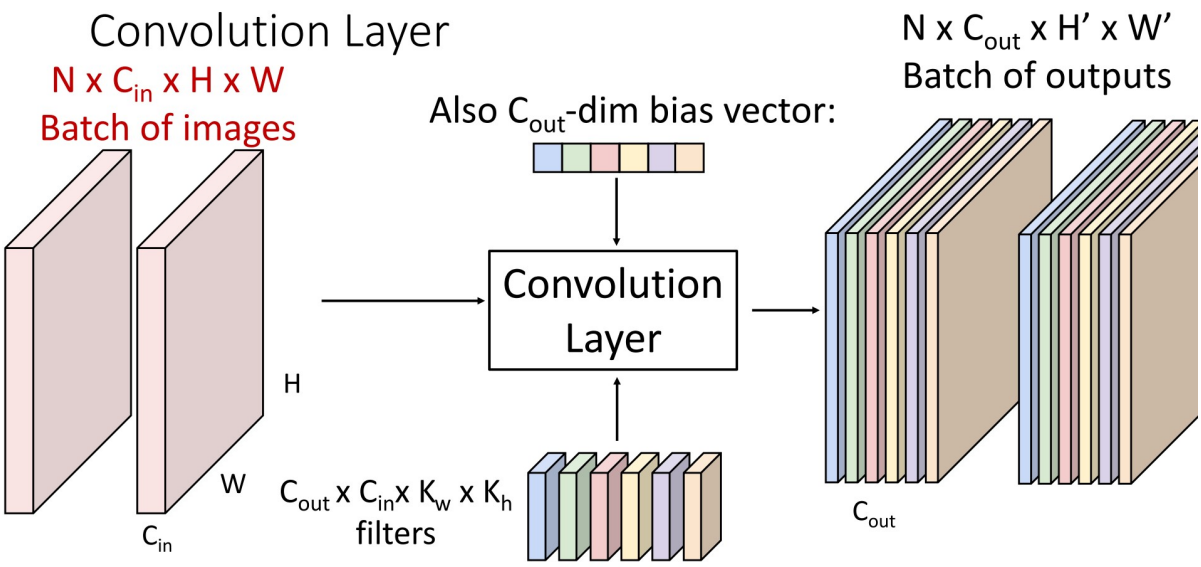


Figure 10: Source: Johnson (2019)

Depth determines how many filters are to be learned and thus defines the dimensionality (C_{out}) of the output volume (the number of activation maps).

Stride determines how the filters are convolved over the input activations, essentially the step size. If the stride is 1, the filter moves one pixel at a time to compute the next activation. If the stride is greater, e.g., 2, it moves two pixels at a time, making the activation maps smaller in width and height.

Padding refers to adding (typically) zeros to the border of the input activations before performing the convolution. This can be useful to ensure, for example, that the spatial dimensions of the activation maps are identical to those of the input activations. This is essential for segmentation tasks. Figure 11 illustrates the problem. Figure 12 shows an example with padding.

Figure 13 shows the interplay between stride and padding.

Dumoulin and Visin (2016) has created some animations for better understanding of convolutions and published them here: https://github.com/vdumoulin/conv_arithmetic.

Calculations

The dimensionality of the activation maps can be calculated using the following formulas:

- i: Side length of the input activations (assumption: square inputs)
- k: Kernel size (assumption: square kernels)

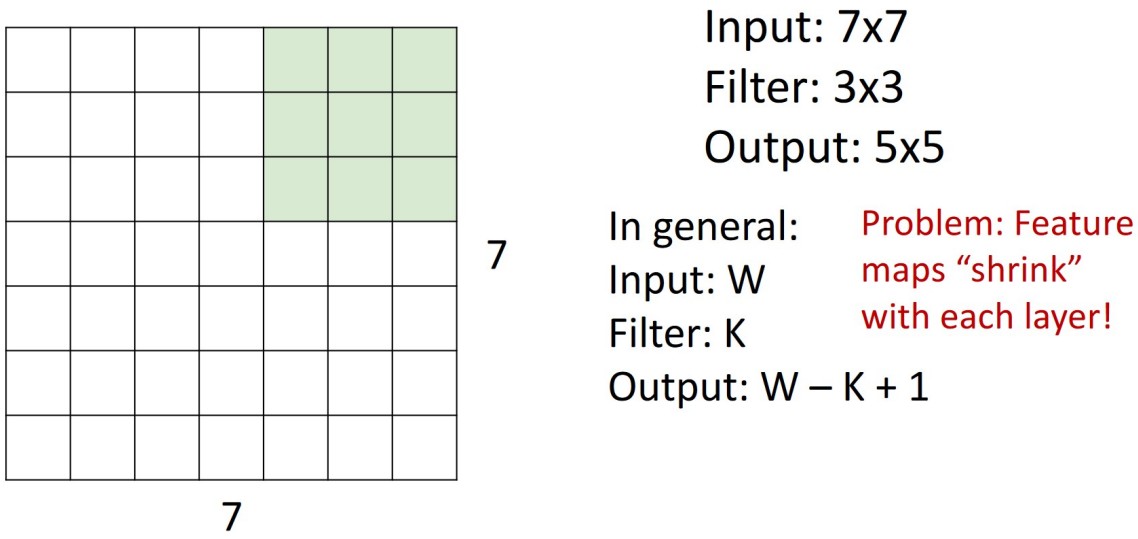


Figure 11: Source: Johnson (2019)

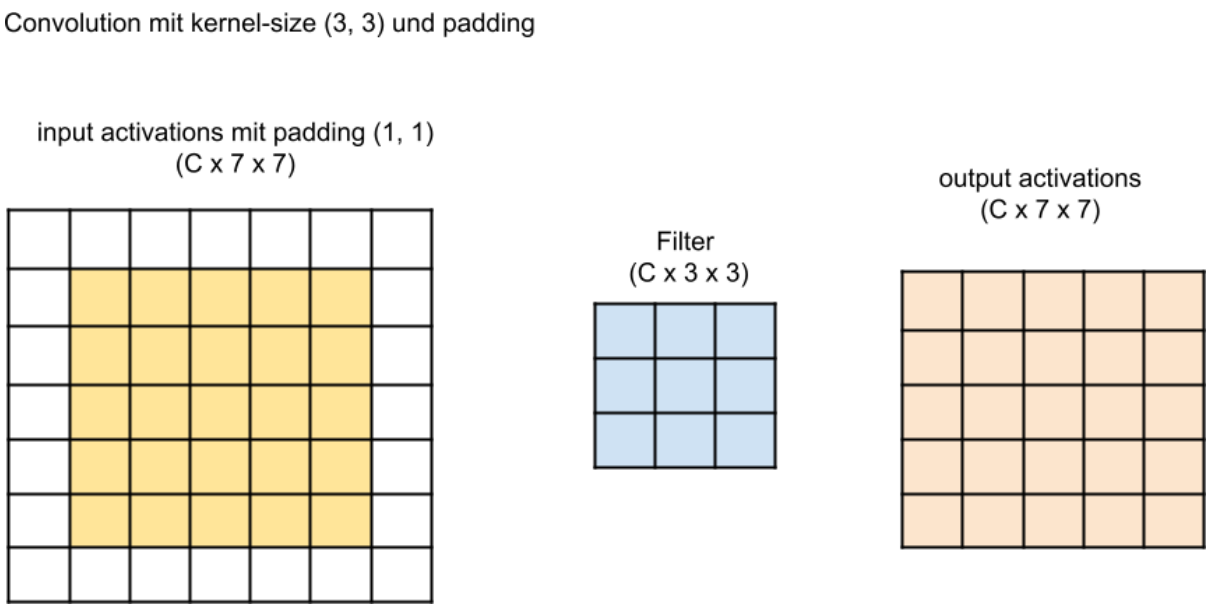


Figure 12: Left: Input (yellow) with zero-padding (1, 1) (white border), middle: Filter, right: Output.

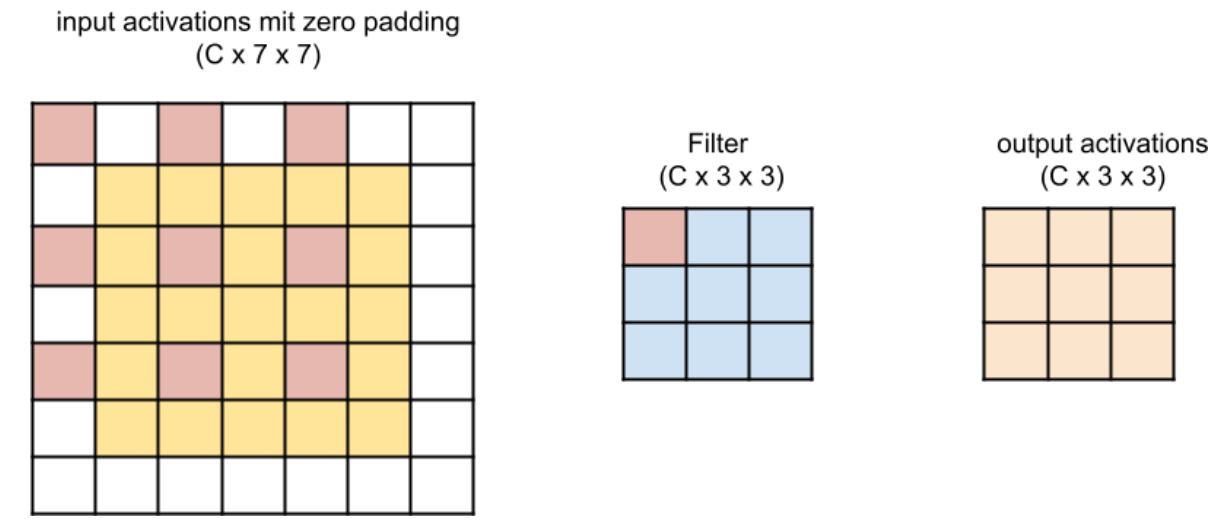


Figure 13: Stride with padding. The red mark indicates the filter value position on the input activations.

- o: Side length of the output activation maps
- s: Stride (assumption: same stride along the spatial dimensions)
- p: Number of paddings on each side (assumption: same number of paddings along the spatial dimensions)

Scenario: *stride* = 1 and without *padding*

$$o = (i - k) + 1 \tag{2}$$

Scenario: *stride* = 1 with *padding*

$$o = (i - k) + 2p + 1 \tag{3}$$

Scenario: Half (same) Padding -> $o = i$

Valid for any i and for odd $k = 2n + 1, n \in N$, $s = 1$ and $p = \lfloor k/2 \rfloor = n$.

$$o = i + 2\lfloor k/2 \rfloor - (k - 1) \tag{4}$$
$$o = i + 2n - 2n \tag{5}$$
$$o = i \tag{6}$$

Scenario: Full Padding

The dimensionality of the output activation can also be increased.

Valid for any i and k, $s = 1$ and $p = k - 1$.

$$o = i + 2(k-1) - (k-1) \tag{7}$$
$$o = i + (k-1) \tag{8}$$

Scenario: No Padding, *stride* > 1

Valid for any i and k, $s > 1$ and $p = 0$.

$$o = \lfloor \frac{i-k}{s} \rfloor + 1 \tag{9}$$

Scenario: *padding* and *stride* > 1

Valid for any i, k, s, p.

$$o = \lfloor \frac{i+2p-k}{s} \rfloor + 1 \tag{10}$$

> **Note**
>
> With this formula, all scenarios are covered!
>
> $$o = \lfloor \frac{i+2p-k}{s} \rfloor + 1 \tag{11}$$

Quiz

> **Question**
> Input: 3 x 32 x 32
> Convolution: 10 filters with 5x5 kernel size, stride=1, and padding=2
> What is the size of the activation map?

> **Question**
> Input: 3 x 32 x 32
> Convolution: 10 filters with 5x5 kernel size, stride=1, and padding=2
> How many weights are there?

Properties

Local (Sparse) Connectivity & Parameter Sharing

Fully connected layers are, as discussed, impractical when working with high-dimensional inputs like images. If all neurons in a layer were connected to all previous neurons, the number of parameters to be estimated would increase massively, which is inefficient and leads to overfitting. Each neuron is therefore only connected to a local region of the input volume. The spatial extent of this region is a hyperparameter and is called the receptive field of a neuron (also kernel size) on the input volume. The connections along the depth (C) extend over the entire depth of the input volume. The connections are therefore local along the spatial dimensions (width and height) but complete along the depth.

Parameter sharing in convolutional layers is used to reduce the number of parameters. Since the filters are convolved over the inputs, the individual weights of the filters are identical over the spatial extent of the input volume. One of the main assumptions behind CNNs is the following: If it is useful to learn a specific (visual) feature at a certain position, then it is probably useful at other positions as well. In other words: If I learn filters that detect edges, corners, or cats, then it is a reasonable assumption that I want to do this throughout the image.

> **Note**
>
> Sometimes parameter sharing does not make sense. This can be the case, for example, if we have centered structures in the images. Then you might want to learn position-dependent features. An example is images of faces that have been centered, where you might want to learn filters that detect the mouth only in the lower middle area (locally connected layers).

The following output shows the number of parameters in an MLP and a CNN (each with two hidden layers) on the CIFAR10 dataset.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchinfo

class MLP(nn.Module):

    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.hidden_layer1 = nn.Linear(3 * 32 * 32, 64)
```

```
        self.hidden_layer2 = nn.Linear(64, 32)
        self.output_layer = nn.Linear(32, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.hidden_layer1(x))
        x = torch.relu(self.hidden_layer2(x))
        x = self.output_layer(x)
        return x

net = MLP()
print(torchinfo.summary(net, input_size=(1, 3, 32, 32)))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MLP                                      [1, 10]                   --
 Flatten: 1-1                            [1, 3072]                 --
 Linear: 1-2                             [1, 64]                   196,672
 Linear: 1-3                             [1, 32]                   2,080
 Linear: 1-4                             [1, 10]                   330
==========================================================================================
Total params: 199,082
Trainable params: 199,082
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.20
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 0.80
Estimated Total Size (MB): 0.81
==========================================================================================
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchinfo

class CNN(nn.Module):

    def __init__(self):
```

```
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 7, stride=2, padding=3)
        self.conv2 = nn.Conv2d(16, 16, 3, stride=2, padding=1)
        self.flatten = nn.Flatten()
        self.output_layer = nn.Linear(16 * 8 * 8 , 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = self.flatten(x)
        x = self.output_layer(x)
        return x

cnn = CNN()
print(torchinfo.summary(cnn, input_size=(1, 3, 32, 32)))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CNN                                      [1, 10]                   --
 Conv2d: 1-1                             [1, 16, 16, 16]           2,368
 Conv2d: 1-2                             [1, 16, 8, 8]             2,320
 Flatten: 1-3                            [1, 1024]                 --
 Linear: 1-4                             [1, 10]                   10,250
==========================================================================================
Total params: 14,938
Trainable params: 14,938
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.76
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.04
Params size (MB): 0.06
Estimated Total Size (MB): 0.11
==========================================================================================
```

Question
How should the linear transformation be defined to obtain the desired result? How many parameters are needed? How could this be done with a convolution?

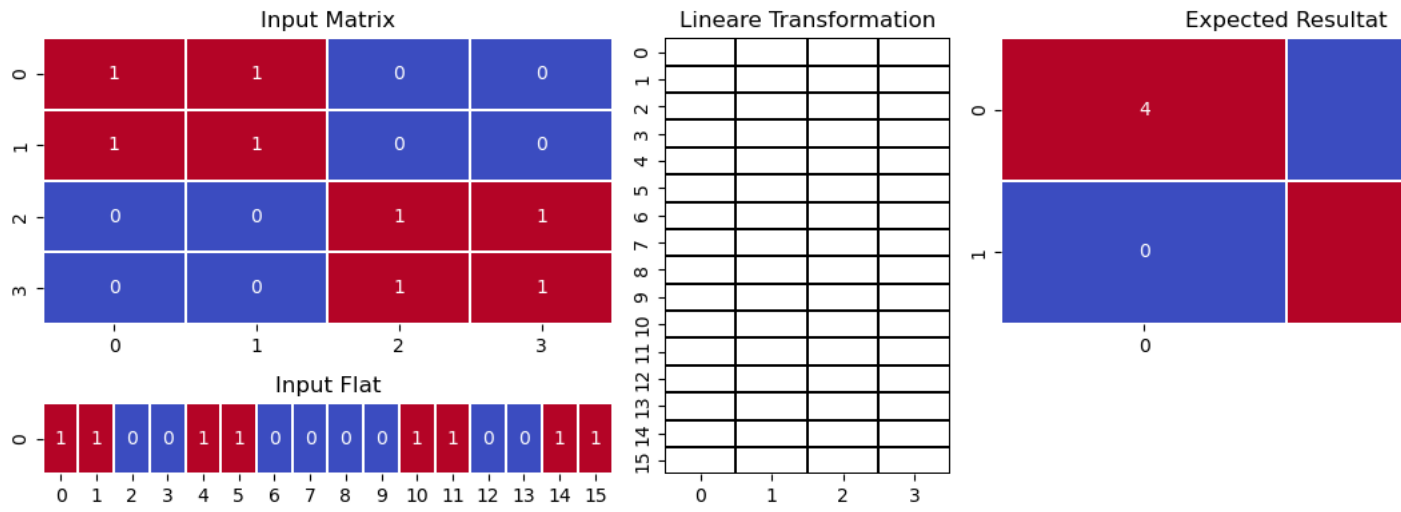


Figure 14: Input in 2-D (top left), the flattened version of it (bottom left), expected output (right), and unknown transformation (middle).

Translation Invariance / Equivariance

Translation invariant is a function that produces the same value under translations $g()$ of the input x:

$$f(g(x)) = f(x) \tag{12}$$

Translation equivariant is a function that produces the same value under translations $g()$ of the input x, provided that it is also shifted by $g()$:

$$f(g(x)) = g(f(x)) \tag{13}$$

Convolutions are translation equivariant, as illustrated well in the following example:

https://www.youtube.com/embed/qoWAFBYOtoU

Stacking & Receptive Field

Multiple convolutions can be executed in sequence (stacking). Each convolution is performed on the activation maps of another previous convolution. Figure 15 illustrates the result.

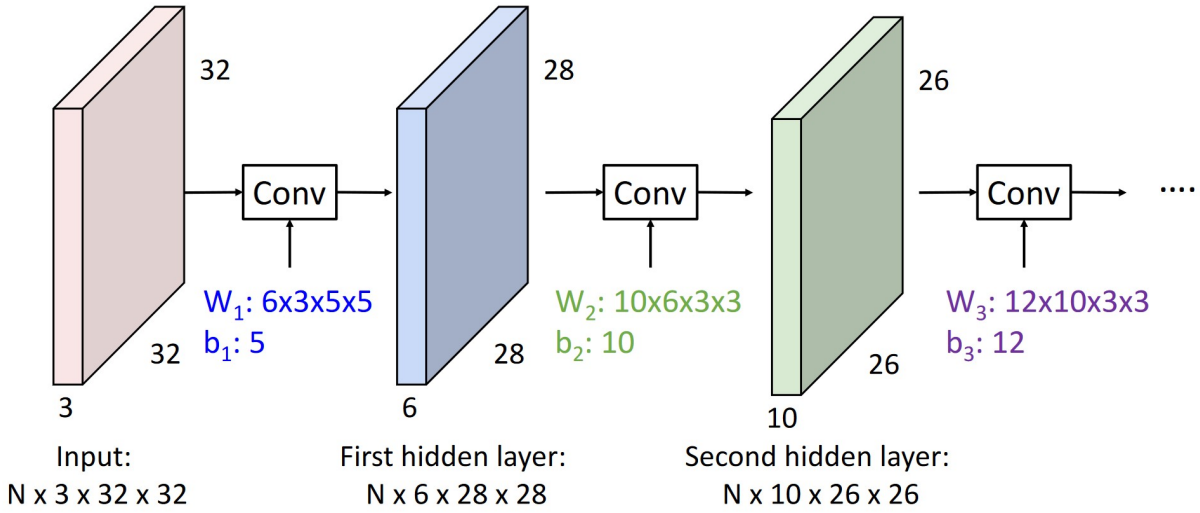


Figure 15: Source: Johnson (2019)

A convolution is therefore not only performed directly on the input (e.g., images) but is generally defined on inputs of dimensionality $H \times W \times C$! (There are also variants in higher dimensions.)

However, non-linear activation functions must be used between the convolutions. Otherwise, the stacked convolution can be expressed with a simple convolution (similar to an MLP, which can be expressed with a linear transformation without activation functions).

The receptive field defines which inputs influence the activations of a neuron. Figure 16 illustrates the concept.

Stacking multiple convolutions increases the receptive field of a neuron relative to the original input (see Figure 17).

Variations

Dilated Convolutions

Dilated convolutions have an additional hyperparameter, the dilation. Dilated convolutions have holes between the rows and columns of a kernel. This increases the receptive field without having to learn more parameters. This variant is also called à trous convolution. **?@fig-cnn-dilation-gif**, Figure 18, and Figure 19 show examples.

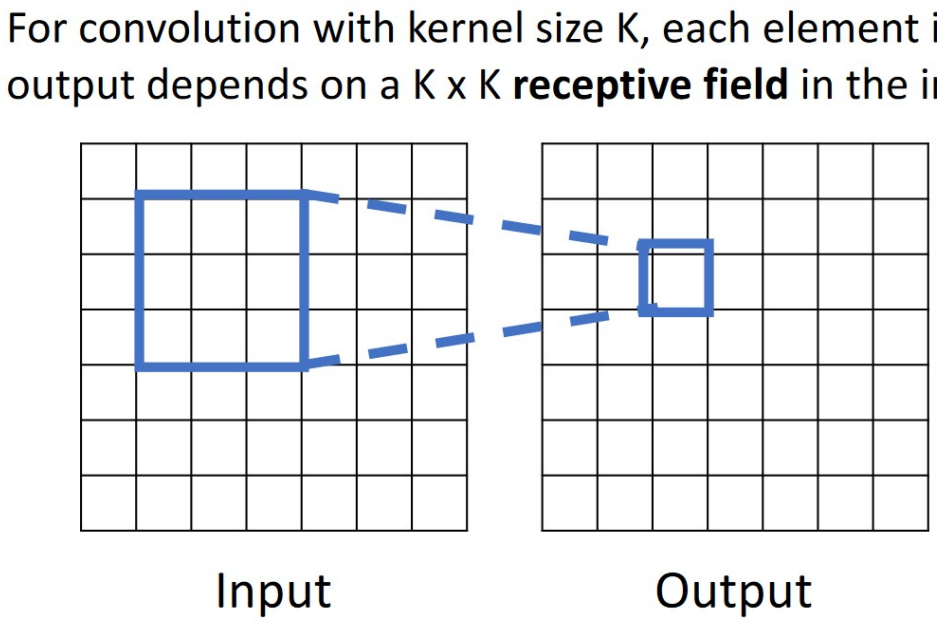


Figure 16: Source: Johnson (2019)

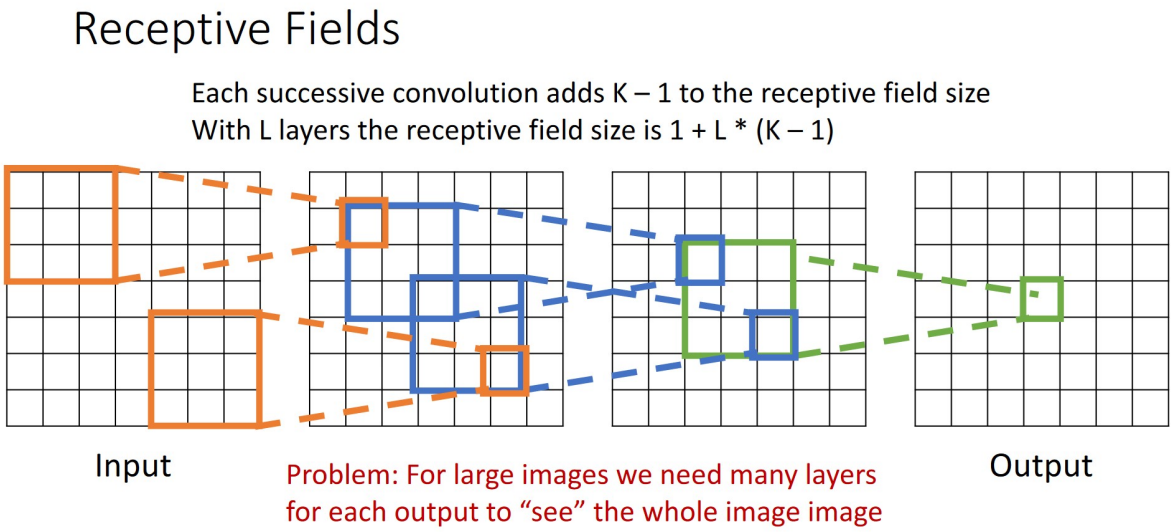


Figure 17: Source: Johnson (2019)

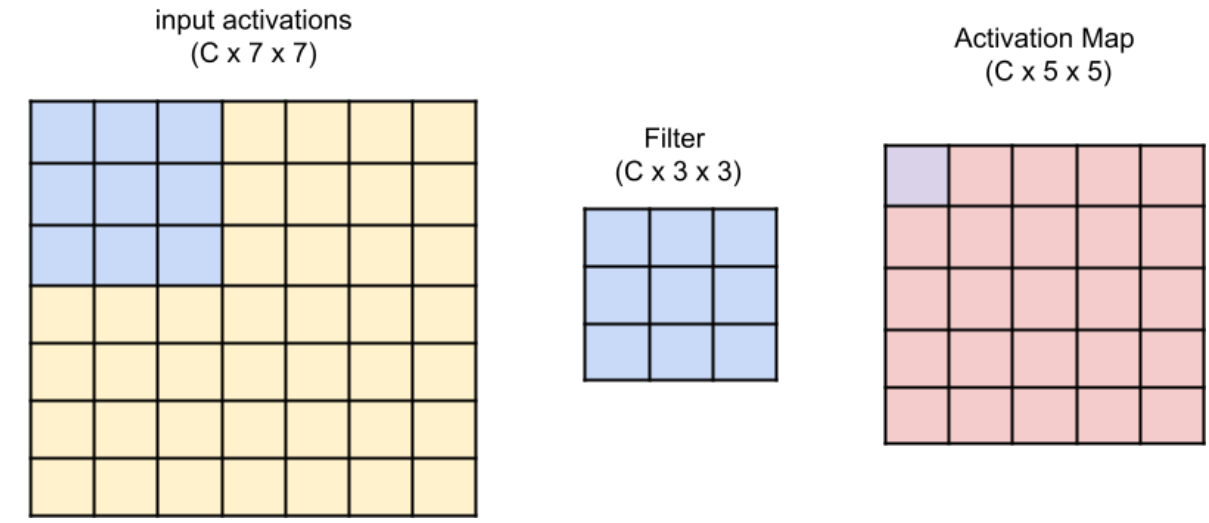


Figure 18: Convolving a 3x3 kernel over a 7x7 input without padding with stride 1x1 and dilation 1.

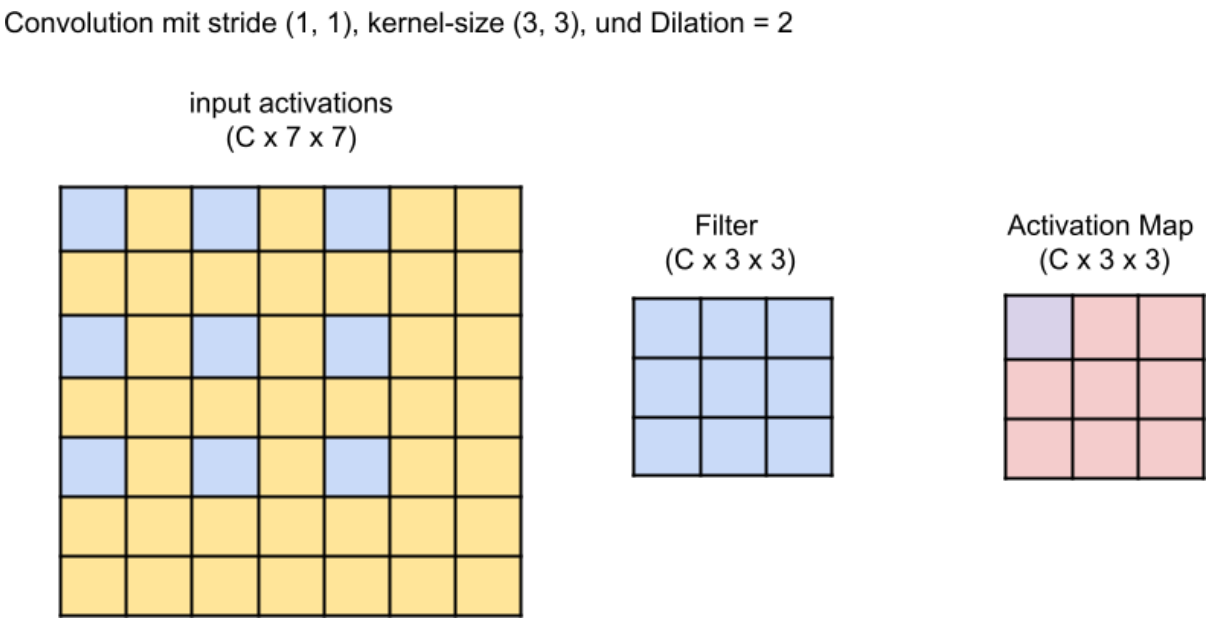


Figure 19: Convolving a 3x3 kernel over a 7x7 input without padding with stride 1x1 and dilation 2.

1x1 (pointwise) Convolutions

1x1 convolutions have a kernel size of 1x1 and thus no spatial extent. These layers are often used in CNNs to change the number (C) of activation maps with few parameters. For example, activation maps of dimensionality ($C \times H \times W$) can be changed to a volume of ($C2 \times H \times W$) with $C2 * (C + 1)$. This can be useful, for example, to save parameters before more complex layers or at the end of the CNN to adjust the size of the activation maps to the number of classes to be modeled (for classification problems) or to reduce to 3 color channels ($C2 = 3$) for image generation models. Figure 20 shows an example.

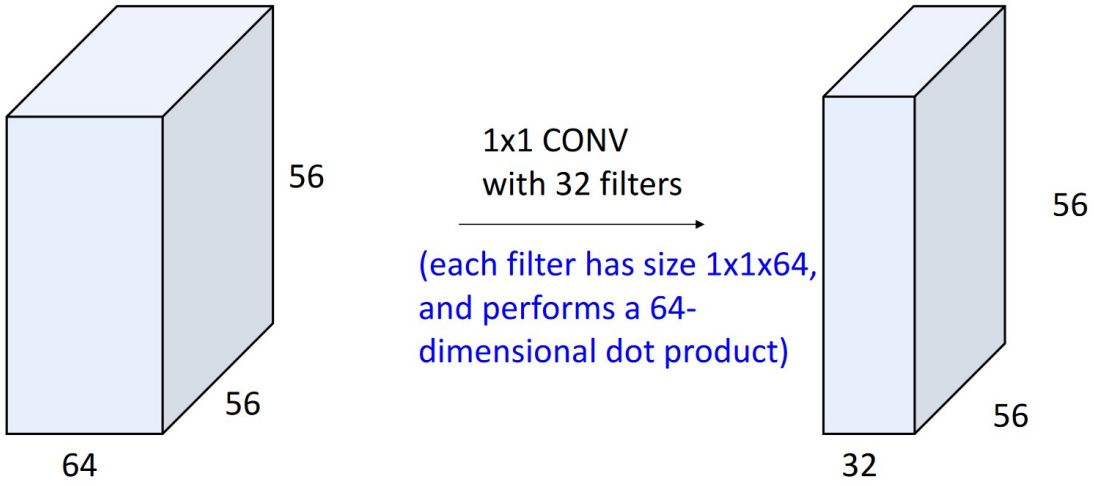


Figure 20: Source: Johnson (2019)

Depthwise Separable Convolutions

Depthwise separable convolutions are a way to further reduce the number of parameters in convolutional layers. Instead of extending filters over the entire depth of the input activations, a separate filter (kernel) is used for each input channel, with the dimensionality ($1 \times K \times K$). Figure 21 shows an example. Subsequently, 1x1 convolutions are used to combine information across the input channels. See Figure 22 for a comparison of 'normal' convolutions and depthwise separable convolutions. Since 1x1 convolutions require fewer parameters, activation maps can be generated with fewer parameters.

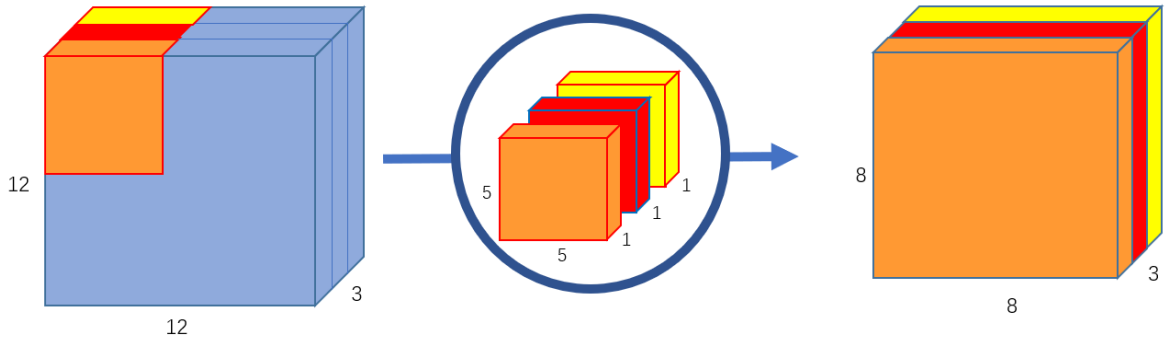


Figure 21: Source: https://paperswithcode.com/method/depthwise-convolution

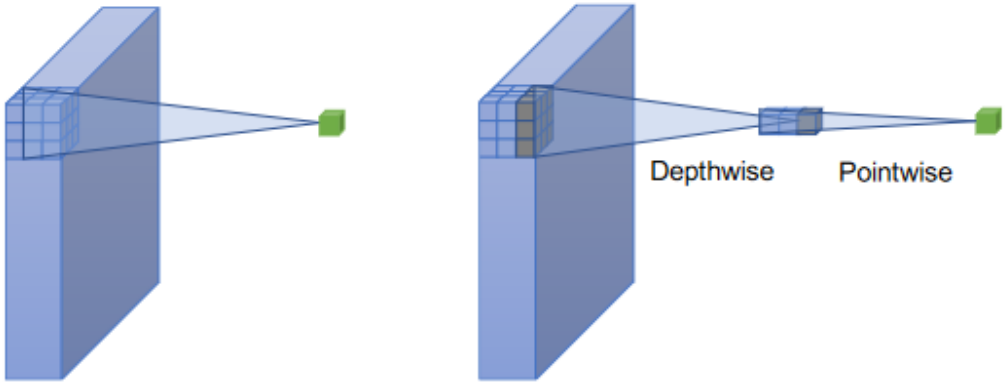


Figure 22: Source: Yu and Koltun (2016)

Pooling Layers

Often, the spatial dimensionality of the activation maps needs to be successively reduced in a CNN. This reduces the number of computations and memory required. Also, information is condensed and aggregated layer by layer: high spatial resolution is often no longer necessary. We have already seen that convolutional layers with a stride > 1 can achieve this goal. However, it is also possible to use pooling layers, which do not have (learnable) parameters.

A frequently used variant is the max-pooling layer. This layer operates independently on each slice along the depth dimension and returns the maximum value. The kernel size and stride must also be defined. A stride of 2×2 with a kernel of 2×2 halves the dimensionality of the activation maps along height and width.

For any i, k, s and without padding:

$$o = \lfloor \frac{i-k}{s} \rfloor + 1 \tag{14}$$

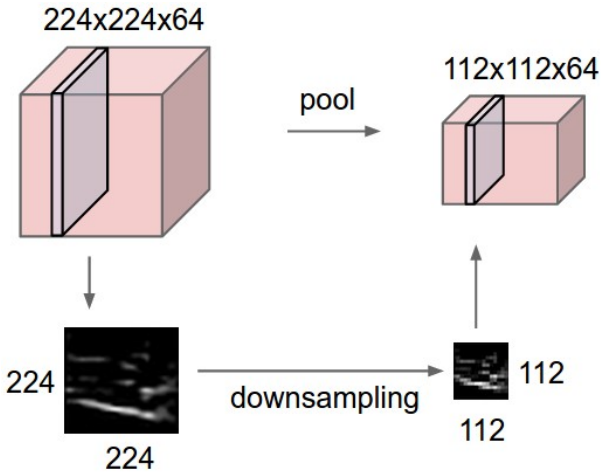


Figure 23: Source: Li (2022)

Figure 27 shows the result of max-pooling, average-pooling and global average-pooling. In average-pooling, instead of choosing the maximum, the average activation is calculated. Otherwise, average-pooling works the same way as max-pooling. A crucial pooling variant is global average pooling. The average of the activations is calculated along the depth dimension (i.e., no kernel size or stride needs to be defined). Activation maps with $(C \times H \times W)$ are reduced to $(C \times 1 \times 1)$. This is useful, for example, to directly model logits in a classification problem with C classes. This allows architectures to be created that do not use fully connected layers at all.

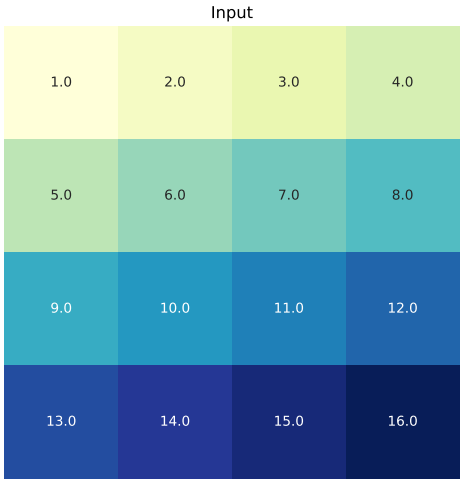


Figure 24: Input

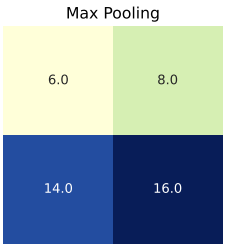


Figure 25: Max Pooling

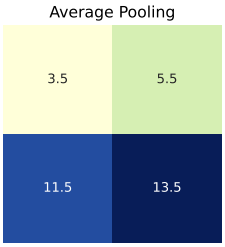


Figure 26: Average Pooling

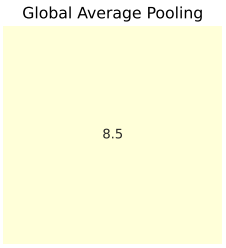


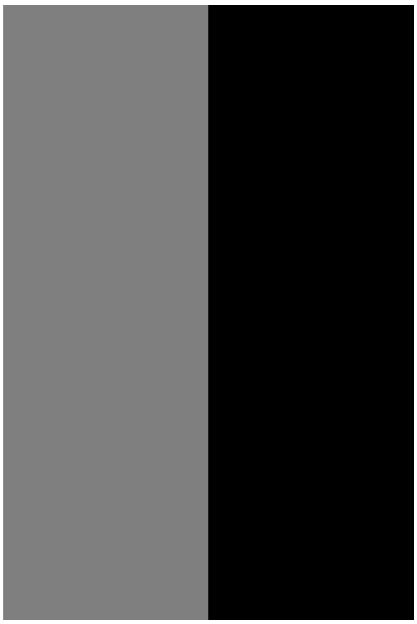
Figure 27: Global Average Pooling

PyTorch Examples

```
import numpy as np
import torch
from torch.nn import functional as F
import torchshow as ts
from PIL import Image
from matplotlib import pyplot as plt
```

```
#img = Image.open({{< meta params.images_path >}}'cat.jpg')
image_path = "../assets/images/cnns/cat.jpg"
img = Image.open(image_path)
img
```

```
filter = torch.tensor(
    [   [[1, 0, -1], [1, 0, -1], [1, 0, -1]], # R
        [[1, 0, -1], [1, 0, -1], [1, 0, -1]], # G
        [[1, 0, -1], [1, 0, -1], [1, 0, -1]], # B
    ]).unsqueeze(0).float()
ts.show(filter, show_axis=False)
```

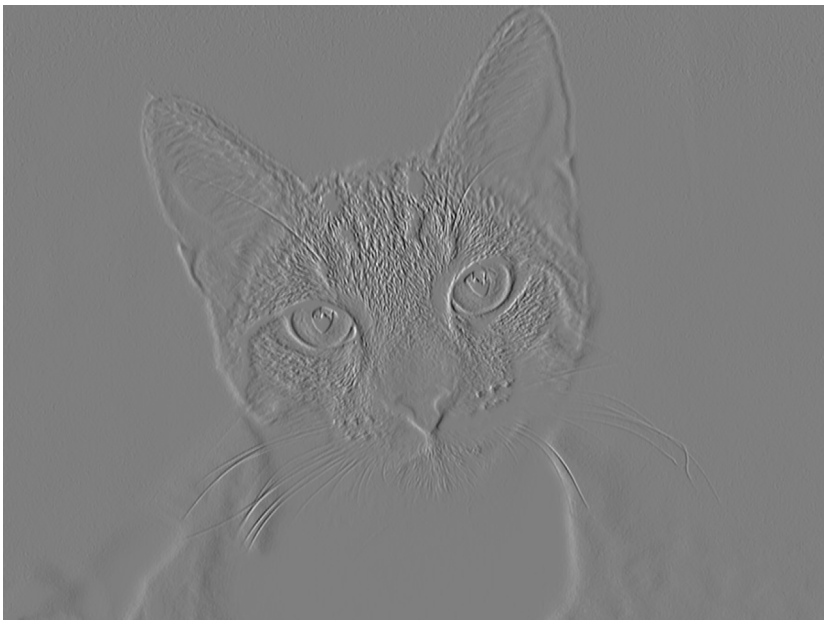


```
input = torch.tensor(np.array(img)).unsqueeze(0).permute(0, 3, 1, 2).float() # (N, C, H, W)
input /= 255.0
input -= 1.0
result = F.conv2d(input, filter, stride=1, padding=0, dilation=1, groups=1)
```

```
ts.show(result)
```

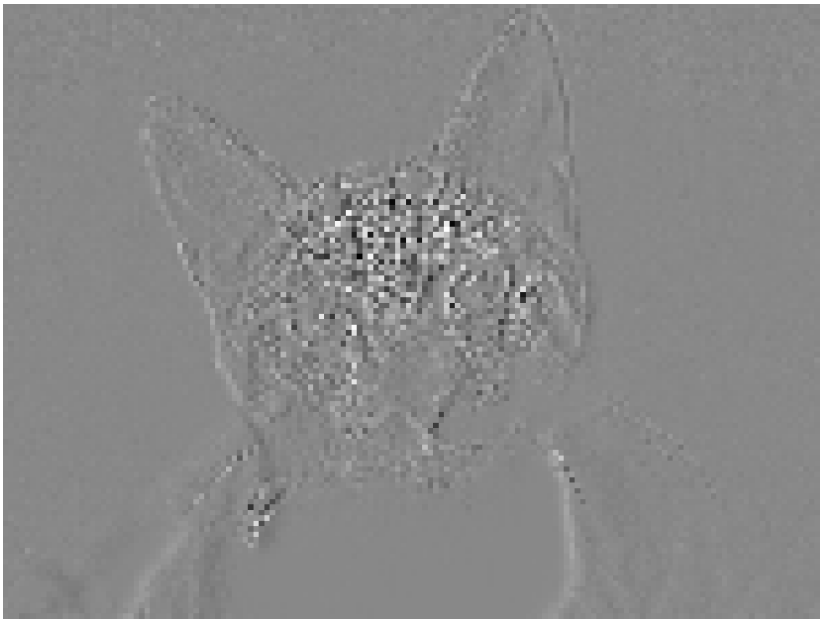
```
/opt/hostedtoolcache/Python/3.11.9/x64/lib/python3.11/site-packages/torchshow/visualization.p

Original input range is not 0-1 when using grayscale mode. Auto-rescaling it to 0-1 by defaul
```



2D-Convolution:

```
result = F.conv2d(input, filter, stride=6, padding=0, dilation=1, groups=1)
ts.show(result)
```



Transposed convolution:

```
result = F.conv2d(input, filter, stride=6, padding=0, dilation=1, groups=1)
result = F.conv_transpose2d(result, weight=torch.ones_like(filter))
ts.show(result)
```

Max-Pooling:

```
result = F.max_pool2d(input, kernel_size=8, stride=8)
ts.show(result)
```

References

Dumoulin, Vincent, and Francesco Visin. 2016. "A Guide to Convolution Arithmetic for Deep Learning." *ArXiv e-Prints*, March.

Johnson, Justin. 2019. "EECS 498-007 / 598-005: Deep Learning for Computer Vision." Lecture {Notes} / {Slides}. https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2019/.

Li, Fei-Fei. 2022. "CS231n Convolutional Neural Networks for Visual Recognition." Lecture {Notes}. https://cs231n.github.io.

———. 2023. "CS231n Convolutional Neural Networks for Visual Recognition." Lecture {Notes}. http://cs231n.stanford.edu/schedule.html.

Yu, Fisher, and Vladlen Koltun. 2016. "Multi-Scale Context Aggregation by Dilated Convolutions." arXiv. http://arxiv.org/abs/1511.07122.